Operating System/2™ Programming Tools and Information Version 1.2

Programming Guide

64F0273

Operating System/2™ Programming Tools and Information Version 1.2

Programming Guide

First Edition (September 1989)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

© Copyright International Business Machines Corporation 1987, 1989. All Rights Reserved.

Note to US Government users - Documentation related to Restricted Rights - Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Special Notices

The following names, used in this publication, are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries:

IBM
IBM Basic Compiler/2
IBM C/2
IBM COBOL/2
IBM FORTRAN/2
IBM Macro Assembler
IBM Pascal Compiler
IBM Personal Computer AT
Operating System/2
OS/2
Personal System/2
PS/2
Presentation Manager
Systems Application Architecture.

The following names, used in this publication, are trademarks or registered trademarks of other companies:

PostScript	Adobe Systems Incorporated
Intel	Intel Corporation
Microsoft Windows	Microsoft Corporation
Teletype	Teletype Corporation

This publication contains sample programs. Permission is hereby granted to copy and store the sample programs into a data processing machine and to use the stored copies for internal study and instruction only. No permission is granted to use the sample programs for any other purpose.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license enquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Armonk, NY 10504.

Preface

About This Book

The *IBM Operating System/2 Programming Guide* gives guidance information and code examples to enable you to start writing source code using many of the calls in the application programming interface (API) of the IBM Operating System/2 Standard Edition Version 1.2 (OS/2). This guide is not a complete and comprehensive reference to the API. For such information, see the *Control Program Programming Reference*. This guide does not tell you how to compile or link your programs. For such information, see the *OS/2 Version 1.2, Building Programs* and the compiler publications for the programming language you are using. This book does not tell you about the Dialog Manager, which is a set of services that provide a high-level programming interface to the Presentation Manager API. This is described in *OS/2 1.2 Dialog Manager Programming Guide and Reference*.

You can read this book sequentially. Alternatively, you may prefer to refer only to those subjects in which you are interested, using the "Contents" on page ix, or the index at the back of the book, to look up the subjects. If you want to get an idea of the scope of the guide at a glance, the structure of the book is shown in "Guide Structure" on page viii.

To illustrate programming with the OS/2 application programming interface (API), this guide makes extensive use of code taken from the sample applications supplied on the OS/2 programming tools and information diskette. The samples are in source code. To install the tools, follow the instructions in the *OS/2 1.2 Programming Tools Installation* card.

The operation of each sample is described in Chapter 2, "OS/2 1.2 Programming Tools." You should familiarize yourself with the operation of each sample from a user's viewpoint. It will help you understand the explanations of the sample code in the guide. Note that the simpler programs contain no error checking, as this would obscure the main programming points. You should, therefore, run the samples with caution.

When you understand the code in the sample applications, you can make copies of them (give the copies names different from the originals). To familiarize yourself with the API, experiment with the calls by changing parameter values in the copies, put in error checking, and observe the effect when you compile and run them. The TEMPLATE sample application is a compilable and startable skeleton C/2 application that you can use as a starting point from which to develop your Presentation Manager C/2 applications for OS/2 1.2.

Prerequisite Knowledge

This book is for application designers and programmers who are familiar with the following:

- The information contained in the *OS/2 Standard Edition Version 1.2, Getting Started, OS/2 Standard Edition Version 1.2, Using Advanced Features* and *OS/2 Version 1.2, Programming Overview* manuals. The *Programming Overview* introduces OS/2 and describes the application types that you can develop using the OS/2 API.
- At least one of the following programming languages in which an OS/2 application can be written:

 IBM Basic Compiler/2
 IBM COBOL/2
 IBM C/2
 IBM FORTRAN/2
 IBM Macro Assembler/2
 IBM Pascal Compiler/2.

Programs that explicitly create windows and share the screen with the OS/2 Presentation Manager can be written only in C/2, Macro Assembler/2, COBOL/2, and FORTRAN/2. The API call names have a long form that you use with C/2 and Macro Assembler/2, (for example, WinCreateWindow), and a short form that you use with COBOL/2 and FORTRAN/2 (for example, WICRT, which is the short form call name of WinCreateWindow). Not all of the Presentation Manager functions are supported in COBOL/2 and FORTRAN/2. There are also a few functions that are supported only in COBOL/2 and FORTRAN/2. In this guide, where a function is supported in C/2, Macro Assembler/2, COBOL/2, and FORTRAN/2, the long name is given, followed by the short name in parentheses,

for example, WinCreateWindow (WICRT). The same convention is used where a function is supported only in COBOL/2 and FORTRAN/2, as such calls are listed under their long names in the reference manuals. Where a function is supported only in C/2 and Macro Assembler/2, the long name only is given.

OS/2 and Related Product Libraries

OS/2 Product

IBM Operating System/2
Standard Edition
Version 1.2

Getting Started
Using Advanced Features
Product Information

6024926 3.5″ diskette
6024930 5.25″ diskette

OS/2 Programming Tools and Information

IBM Operating System/2
Version 1.2

Installation
Programming Overview
Programming Guide
Building Programs
Dialog Tag Language
Guide and Reference
Dialog Manager Guide
and Reference
Dialog Manager and
Dialog Tag Language
Reference Summary
Programming Reference
(3 books)
Bindings Reference for
Presentation Manager
(4 books for COBOL/2,
FORTRAN/2, C/2, and
Macro Assembler/2)
I/O Subsystems and
Device Support
(2 books)
Systems Application
Architecture
Common User Access:
Advanced Interface
Design Guide

6024929

Separate Order (no charge)

Keyboards and Code Pages

6280345

Available Separately

Command Reference

6024928

Service Coordinator's
Guide

15F2214

Programming Languages

IBM Basic Compiler/2
Version 1.0 6280179

IBM Macro Assembler/2
Version 1.0 6280181

IBM Pascal Compiler/2
Version 1.0 6280183

IBM FORTRAN/2
Version 1.02 6280185

IBM COBOL/2
Version 1.0 6280207

IBM C/2
Version 1.1 6280284

Systems Application Architecture

An Overview

GC26-4341

Writing Applications:
A Design Guide

SC26-4362

Common User Access:
Advanced Interface
Design Guide
SC26-4582

Common User Access:
Basic Interface Design
Guide
SC26-4583

Common Programming Interface

C Reference - Level 2

SC09-1308

COBOL Reference

SC26-4354

Dialog Reference

SC26-4356

FORTRAN Reference

SC26-4357

Procedures Language
Reference
SC26-4358

Presentation Reference

SC26-4359

Guide Structure

Parts 2 through 6 of this guide describe the OS/2 Presentation Manager functions for programming a user interface for a display screen, and a mouse or keyboard. You could collectively term the functions as direct service to the user.

Part 7 describes the OS/2 Control Program functions, most of which can be used with any of the Presentation Manager or Dialog Manager functions. The Control Program functions are usually internal to applications. You could collectively term the functions as service to the application.

Part 8 describes support for hard-copy output devices.

Part 9 describes various product development topics.

Note that this guide does not describe the Dialog Manager. The Dialog Manager provides a set of services that are a higher-level programming interface to the presentation management functions. Being a higher level interface, the Dialog Manager provides a tag-language-based method of specifying panels, and a programming interface that calls Dialog Manager services to display and manage the panels, but does not allow your program the same degree of control of the user interface. The Dialog Manager is described in the *OS/2 1.2 Dialog Manager Programming Guide and Reference* manual.

The "Contents" on page ix lists the contents of the guide in detail. In brief, the topics described by each part are as follows:

"Part 1. Introduction"
Introduces the IBM Operating System/2 application types, and how the OS/2 API and programming tools support them.

"Part 2. Presentation Manager Applications – Using the Windowed Programming Interface"
Describes how to use windows for your application user interface.

"Part 3. Presentation Manager Applications – Using OS/2 Information Presentation Facility"
Describes the Help Manager, a programming tool that a Presentation Manager application can call, to manage the display of help panels in response to user help requests.

"Part 4. Presentation Manager Applications – Using the Graphics Programming Interface"
Describes how to draw graphics pictures in a windowed application.

"Part 5. Presentation Manager Applications – Data Transfer Between Running Applications"
Describes how to use two similar methods – Clipboard Data Exchange and Dynamic Data Exchange – for transferring data between running windowed applications.

"Part 6. Alphanumeric Text Applications – User Input/Output"
Describes how to use fixed-size alphanumeric text in Presentation Manager, text-windowed, and full-screen applications.

"Part 7. Control Program Functions"
Describes file I/O, multitasking, interprocess communication, and memory management.

"Part 8. Printing and Device Support"
Describes printing and device support.

"Part 9. Product Development Considerations"
Describes facilities that assist the development of program products, such as national language support, and migration of existing programs.

Contents

Part 1. Introduction

Chapter 1. Introduction to the IBM Operating System/2 Application Programming Interface 1-1
OS/2-Mode and DOS-Mode 1-1
OS/2-Mode and DOS-Mode Application Types 1-2
OS/2-Mode Applications that Run in Windows 1-2
Full-Screen Applications 1-4
DOS-Mode Applications 1-4
How the OS/2 Application Classes Run Concurrently 1-4
OS/2 Mode and How It Breaks the 640KB Barrier 1-6
How an Application Uses the OS/2 Application Programming Interface 1-6
Creating Your Own Dynamic Link Libraries 1-6
What is in the OS/2 1.2 Application Programming Interface? 1-6
The API Calls Used By Windowed Applications 1-7
The API Calls Used By Text-Windowed Applications 1-7
The API Calls Used By Full-Screen Applications 1-8
The API Calls Used By DOS-Mode Applications 1-8
Language Support 1-8
386 Programming Considerations 1-9

Chapter 2. OS/2 1.2 Programming Tools . . 2-1
Include and Header Files 2-1
C/2 and Macro Assembler Files 2-1
COBOL/2 and FORTRAN/2 Include Files 2-2
COBOL/2 and FORTRAN/2 Structures . 2-3
The PM Sample Programs in C/2 2-4
The "Hello" C Sample – HELLO1 2-4
The Standard Window C Sample – HELLO2 2-4
The Dialog Box Introductory C Sample – DIALOG1 2-4
The Dialog Boxes and Controls C Sample – DIALOG2 2-4
The Retained Graphics C Sample – GRAPHIC1 2-6
The Non-Retained Graphics C Sample – GRAPHIC2 2-6
The Clipboard C Sample – CLIPBRD . . 2-7
The Image C Sample – IMAGE 2-8
The Bit Map C Sample – BMAP 2-9
The Advanced Vio C Sample – AVIOSAMP 2-9
The Typing C Sample – TYPETEXT . . 2-10
The Font Test C Sample – FONTTEST 2-11
The Template C Sample – TEMPLATE 2-11
The Hello World Macro Assembler/2 Sample – HELLOM 2-12
The PM Sample Programs in COBOL and FORTRAN 2-12
Control Program Sample Programs 2-12
Control Program Message Utilities 2-13
Presentation Manager Programming Utilities 2-13
Help Manager Utilities 2-14

Part 2. Presentation Manager Applications – Using the Windowed Programming Interface

Chapter 3. Windows – Concepts and Fundamentals 3-1
Sharing and Subdividing the Screen 3-1
Window Hierarchies 3-1
Window Relationships 3-1
Window Coordinates 3-2
Input to Windows 3-3
Messages 3-3
Structure of PM Applications Written in a Reentrant Language 3-6
The Input Queue 3-6
Posting and Sending Messages 3-7
Input Queue Function Calls 3-7
The Window Procedure 3-8
Programming without Application Window Procedures 3-8
Mixed Language Programs 3-12
Window Classes and Object-Oriented Programming 3-12
Window Subclassing 3-14
Creating Windows 3-14
Window Styles 3-15
The Input Process 3-15
User Responsiveness 3-18
Output in Windows 3-18
The Presentation Space 3-18

Visible Region 3-19
Update Region 3-19
Synchronous and Asynchronous Painting 3-20
Device Contexts 3-21
Conforming to Common User Access 3-22
Primary, Secondary, and Pop-up Windows 3-23
Limits for Number of Windows 3-24

Chapter 4. The HELLO1 Sample Application 4-1
Example Code: HELLO1 Sample PM Application 4-2
HELLO1 Sample Application Walk-Through 4-4
Include Files 4-4
Function Prototypes 4-4
The Main Function 4-4
Creating a Message Queue 4-5
Registering a Window Class 4-5
Creating a Window 4-5
Positioning and Activating a Window . . . 4-6
The Message Loop 4-7
Application Termination 4-7
The Window Procedure 4-7
The WM_PAINT Message 4-8
The WM_BUTTON1DOWN Message . . 4-8
The WM_CHAR Message 4-8
The WM_CLOSE Message 4-9
Default Processing of Messages 4-9
Window Procedure Recursion 4-9
Nonreentrant Equivalent of HELLO1 4-10
Include Files 4-10
Function Prototypes 4-10
The Main Function 4-10
The Window Routine 4-11
Traditional Equivalent of HELLO1 4-11

Chapter 5. The Standard Window 5-1
Window Ownership 5-2
Sending and Posting Messages to a Window 5-2
Creating a Standard Window 5-3
Example Code: Creating a Standard Window, Containing a Client Window . 5-3
Passing Control Data and Presentation Parameters to a Newly-Created Window 5-6
Frame Window Resources 5-6
Example Code: Frame Window Resources 5-7
Standard Window Visual Components 5-7
Title Bar 5-7
Maximize/Minimize/Restore Buttons . . . 5-8
Action Bar and Associated Pull-Downs . . 5-8
System Menu Icon 5-15
Sizing Borders 5-15
Scroll Bars 5-16
Client Window 5-19
Adding Controls to the Standard Window . . 5-20

Chapter 6. The HELLO2 Sample Application 6-1
The Resource Script File 6-1
Example Code: HELLO2 Sample PM Application 6-4
Additional Include Files 6-7
Declarations of Character Variables 6-7
Changes to the Main Procedure 6-7
Changes to the Window Procedure 6-8
The WM_CREATE Message 6-8
The WM_SETFOCUS Message 6-8
The WM_COMMAND Message 6-8
The WM_ERASEBACKGROUND Message 6-9
The WM_PAINT Message 6-9
The WM_CLOSE Message 6-9
HELLO2 Sample PM Application – Conclusion 6-9

Chapter 7. Dialog Boxes and Controls 7-1
Message Communication Between a Dialog Box and Its Controls 7-1
Dialog Processing in a Reentrant Language . 7-1
Dialog Processing in a Nonreentrant Language 7-2
Modeless and Modal Dialogs 7-2
Modeless Dialog 7-2
Modal Dialog 7-4
Passing Control Data to a Newly-Created Dialog Box 7-6
Sending and Posting a Control Message to a Control Window 7-6
Setting Text in a Control Window 7-6
Control Windows 7-7
Single Choice – Radio Buttons 7-7
Multiple Choice – Check Boxes 7-8
Immediate Action – Pushbuttons 7-8
Other Button Styles 7-9
Receiving Button Notification Messages . 7-9
Sending and Posting Button Control Messages 7-9
Scrollable Selection Fields – List Boxes . . . 7-10
Receiving List Box Notification Messages 7-11
Sending and Posting List Box Control Messages 7-11
Text Entry – Entry Fields 7-12
Single Line Entry Fields 7-12
Receiving Single Line Entry Field Notification Messages 7-13
Sending and Posting Single Line Entry Field Control Messages 7-13
Multiline Entry Fields 7-14
Receiving Multiline Entry Field Notification Messages 7-15
Sending and Posting Multiline Entry Field Control Messages 7-15
Scrolling Increment for Arrow Keys in Entry Fields 7-17
Combination Boxes 7-17

Receiving Combination Box Notification Messages 7-18
Sending and Posting Combination Box Control Messages 7-18
Scrolling Panel Contents – Scroll Bars . . . 7-18
Displaying Errors – Message Boxes 7-19
Message Box Pushbuttons 7-19
Message Box Icons 7-20
Movable Message Boxes 7-20
Message Box Modality 7-20
Static Controls 7-21
The Dialog Box Editor 7-21
Dialog Box Layout 7-21
Creating a Dialog Box 7-22
Setting a Grid – Grid Option 7-23
Entering Controls 7-23
Testing the Dialog Box 7-25
Group Control Ordering – Order Groups 7-25
Edit an Existing Dialog Box 7-25
Ending an Edit Session 7-25
The Dialog Template 7-26
Using Controls Without Dialog Boxes 7-28
Creating Your Own Controls 7-28

Chapter 8. Resources and Resource Files . . . 8-1
The Icon Editor 8-1
Associating an Icon with a Text-Windowed or Full-Screen Application 8-4
Text-String Tables 8-4
Help Panels and Tables 8-5
Writing a Resource Script File 8-5
Example Code: Skeleton Resource Script File 8-5
Creating a Resource File and Binding It to Your .EXE File 8-6
Method 1 – Putting All the Resources in the .RC File 8-6
Method 2 – Putting All the Resources Except Dialog Boxes in the .RC File . . 8-7
Comparing Method 1 with Method 2 . . . 8-7
The Font Editor 8-8
Editing Functions 8-8
Font Resource File 8-9
Putting a Resource File in a Dynamic Link Library Module 8-10

Chapter 9. DIALOG1 Dialog Box Sample Application 9-1
Resource Script File 9-2
Dialog Template 9-3
Dialog Include File 9-3
Example Code: DIALOG1 Sample Application 9-4
DIALOG1 Window Procedure MyWindowProc 9-8
The WM_CREATE Message 9-8
The WM_COMMAND Message 9-8
The WM_PAINT Message 9-8
The WM_CLOSE Message 9-8
DIALOG1 Dialog Procedure MyDlgProc . . 9-8
DIALOG1 Module Definition File 9-9

Chapter 10. The DIALOG2 Sample Application 10-1
The Main Function 10-1
Example Code: fnwpClient Window Procedure 10-1
The Options Pull-Down 10-3
The WM_CLOSE Message 10-5
The Dialog Procedures 10-5
Example Code: Entry Field Dialog Template 10-5
Example Code: fnwpEntryFieldDlg Entry Field Dialog Procedure 10-6
The WM_COMMAND Message 10-7
Example Code: Auto Radio Button Dialog Template 10-8
Example Code: fnwpAutoRadioButtonDlg Radio Button Dialog Procedure 10-8
The WM_INITDLG Message 10-10
The WM_CONTROL Message 10-10
The WM_COMMAND Message 10-10
Example Code: Check Box Dialog Template 10-10
Example Code: fnwpCheckBoxDlg Check Box Dialog Procedure 10-11
The WM_INITDLG Message 10-13
The WM_CONTROL Message 10-13
The WM_COMMAND Message 10-13
Example Code: CheckColor Private Function 10-13
Example Code: List Box Dialog Template 10-14
Example Code: fnwpListBoxDlg List Box Dialog Procedure 10-15
The WM_INITDLG Message 10-17
The WM_CONTROL Message 10-17
The WM_COMMAND Message 10-17
Example Code: fnwpComboBoxDlg Combobox Dialog Procedure 10-17
The WM_INITDLG Message 10-19
The WM_CONTROL Message 10-19
The WM_COMMAND Message 10-19
Example Code: fnwpModelessDlg Modeless Dialog Procedure 10-20
The WM_INITDLG Message 10-22
The WM_ADJUSTWINDOWPOS Message 10-22
The WM_CONTROL Message 10-22
The WM_COMMAND Message 10-22

Chapter 11. The DLG1 Nonreentrant Sample Application 11-1

The DLG1COB.COB Dialog Box Sample Application 11-1
Example Code: Dialog Resource Script File 11-2
Example Code: Dialog Template File . . . 11-3
Example Code: Dialog Include File 11-3
COBOL Program Structure 11-4
Example Code: DLG1COB Data Division 11-4
DLG1COB Data Division Walk Through . . 11-6
Including Constants 11-6
Variables and Structures 11-6
Example Code: DLG1COB Main Procedure 11-7
DLG1COB Main Procedure Walk Through 11-10
Initialization 11-10
Initializing PM 4-Byte Integers 11-11
Window Creation 11-11
The Message-Processing Loop 11-12
Application Termination 11-12
Example Code: DLG1COB Client Window Routine 11-12
DLG1COB Client Window Routine Walk Through 11-15
The WM-COMMAND Message 11-15
The WM-PPAINT Message 11-16
The WM-CLOSE Message 11-16
Default Window Processing 11-17
Example Code: DLG1COB Dialog Routine 11-17
DLG1COB Dialog Routine Walkthrough . 11-19
The WM-COMMAND Message 11-19
Default Dialog Processing 11-19
Modifying and Querying PM 4-Byte Integers 11-19
Bit-Management Call Summary 11-20

Chapter 12. The DLG2 Nonreentrant Sample Application 12-1
DLG2COB Data Division Walk Through . . 12-1
DLG2COB Main Procedure Walk Through . 12-1
Example Code: MyClientWindowRoutine 12-2
DLG2COB Client Window Routine Walk Through 12-4
The WM-PAINT Message 12-4
The WM-HELP Message 12-4
The WM-CLOSE Message 12-4
The WM-COMMAND Message 12-4
Sample Code: Dialogs pull-down in the .RC file 12-4
Example Code: ClientCommand Routine . 12-5
ClientCommand Routine Walk Through . 12-10
The MI-EXIT Command Value 12-10
The MI-RESUME Command Value . . 12-10
The MI-ENTRYFIELD Command Value 12-10
The MI-AUTORADIOBUTTON Command Value 12-11
Example Code: Auto Radio Button Dialog Template 12-11
The MI-CHECKBOX Command Value 12-12
Example Code: Check Box Dialog Template 12-12
The MI-LISTBOX Command Value . . 12-13
Example Text: List Box Dialog Template 12-13
Example Code: Initialize Listbox Routine 12-14
Dialog Routine MyAutoRadioButtonDlgRoutine . . . 12-15
MyAutoRadioButtonDlgRoutine Walk Through 12-18
The WM-HELP Message 12-18
The WM-PCONTROL Message 12-18
The WM-COMMAND Message 12-19
Default Dialog Processing 12-19
Dialog Routine MyCheckBoxDlgRoutine 12-19
MyCheckBoxDlgRoutine Walk Through . 12-22
The WM-HELP Message 12-22
The WM-PCONTROL Message 12-22
CheckColor Routine 12-23
CheckColor Routine Walk Through 12-24
The WM-COMMAND Message 12-25
Default Dialog Processing 12-25
Dialog Routine MyEntryFieldDlgRoutine 12-25
MyEntryFieldDlgRoutine Walk Through . 12-28
The WM-HELP Message 12-28
The WM-COMMAND Message 12-28
IsIntInRange Routine 12-28
IsIntInRange Walk Through 12-29
Default Dialog Processing 12-30
Dialog Routine MyListBoxDlgRoutine . 12-30
MyListBoxDlgRoutine Walk Through . . . 12-34
The WM-HELP Message 12-34
The WM-PCONTROL Message 12-34
The WM-COMMAND Message 12-34
Default Dialog Processing 12-35
Using Data Structures 12-35
Structures in Structures 12-36
Creating Structures 12-36
Destroying Structures 12-38
Summary of Structure-Handling Calls . 12-38

Chapter 13. Additional Windowing Techniques 13-1
Presentation Manager Limits 13-1
Recognizing and Recovering from Limits 13-2
Using Presentation Parameters 13-2
Presentation Parameter Function Calls . . 13-3
Defining Your Own Messages 13-3
Registering Data Types and Messages . . 13-4
Example Code: Using External Application-Defined Messages 13-5
Registering a Public Window Class 13-9
Registering a Private Window Class 13-9
Subclassing a Window 13-9
Example Code: Subclassing an Instance of a Window 13-10
Creating an Object Window 13-14

Using Window Words 13-14
Example Code: Window Words 13-15
Window Words Function Calls 13-18
Controlling the State of the Keyboard . . . 13-18
Hooks 13-19
Example Code: Using a Help Hook . . . 13-20
Hooks Function Calls 13-21
The Atom Manager 13-21
The Atom Table 13-21
Integer Atoms 13-21
Using Atoms 13-22
Atom Management Function Calls . . . 13-22
The Heap Manager 13-22
Moveable Heaps 13-23
Why the Heap Manager Exists 13-24

Chapter 14. Customizing the User Interface . 14-1
The Initialization Files 14-1
Initialization for Different Environments . 14-1
Accessing Initialization Information 14-1
Function Calls for Initialization Files . . . 14-2
The Installed Program List 14-3
Installed-Program Function Calls 14-4
Starting and Terminating Installed Applications 14-4
List of Running Programs 14-4
Switch List Function Calls 14-5
Saving the Current State of Your Application 14-5
Querying System Values 14-5
Using the System Colors 14-7
Programmer's Responsibility 14-7

Part 3. Presentation Manager Applications — Using OS/2 Information Presentation Facility

Chapter 15. An Introduction to the OS/2 Information Presentation Facility 15-1
The User Requirement for Online Help . . . 15-1
Access to Help from the Application Interface 15-1
Developing a Help Interface 15-1
Coding the Application 15-2
Creating a Library of Help Panels 15-2
A User's Perspective of the Help Interface . . 15-2
Help Window Characteristics 15-2
Help Action Bar 15-3

Chapter 16. Developing Application Code for the Help Interface 16-1
Creating a Help Instance 16-1
Initializing the HELPINIT Structure . . . 16-1
Associating the Instance with the Window Chain 16-2
Allowing IPF Access to the Active Window 16-2
Ending the Help Instance 16-2
Example Code: HELP1 Main Procedure . . . 16-3
Processing Help Requests 16-5
Help for Help Request 16-5
Keys Help Request 16-5
When No Help Is Available 16-5
IPF Messages 16-6
Example Code: HELP1 Window Procedure . 16-7
Help Tables Used by IPF 16-10
Defining Help Tables in Memory 16-10
Defining Help Tables as Resources . . . 16-11
Defining Window Resources 16-11
Help Pull-Down 16-11
Help Pushbutton 16-12
Defining a Command Entry Field 16-12
A Customized Action Bar 16-12
Creating More Than One Help Instance . . 16-13
Installing an Application-Defined Help Hook 16-13
IPF Function Calls 16-14
IPF Error Handling 16-14
Return Value from a Call or Message . . 16-14
HM_ERROR Message 16-14

Chapter 17. Developing Help Information . . 17-1
The Information Developer's Perspective of the Help Interface 17-1
Planning the Help Interface 17-1
IPF Tag Language Markup 17-1
Defining a Panel with IPF Markup 17-2
Headings — The :h Tags 17-2
Index Entries 17-3
Source File Structure 17-4
The :userdoc. Tag 17-4
Heading Sequences 17-4
Imbedded Files — .im 17-5
Naming Source Files 17-5
Commenting Files — .* 17-5
Text Presentation 17-6
Highlighting Text — The :hp. Tags 17-6
Notes 17-6
Notices — The :caution. and :warning. Tags 17-7
Lists 17-7
Illustrating Text 17-11
Controlling Line Formatting 17-12
Creating Special Characters 17-12
Graphic Illustration 17-12
The :artwork. Tag 17-12
Linking from Hypertext Phrases — The :link. Tag 17-13
Linking to Another Panel 17-13
Linking to a Smaller Window 17-14

Sending a Message to the Application . 17-14
Starting Another Presentation Manager Application 17-16
Linking from Hypergraphic Areas 17-16
Link Definition File — The :artlink. Tag 17-17
Compiling Source Files 17-17
Dialog Manager Support 17-18
National Language Support (NLS) . . . 17-18
Error Message File 17-18
Location of .HLP Libraries 17-19
A Reminder 17-19
Keys Help 17-19
Help for Help 17-19

Part 4. Presentation Manager Applications — Using the Graphics Programming Interface

Chapter 18. An Introduction to the Graphics Programming Interface 18-1
The GPI Query Calls 18-2
The Sample Programs 18-2
Program Structure 18-2

Chapter 19. The Graphics Presentation Space 19-1
Defining the Presentation Space 19-1
The Device Context 19-2
Associating a Presentation Space with a Device Context 19-2
The Presentation Page 19-3
Defining the Presentation Page 19-3
Mapping the Presentation Page to the Device 19-5
Cached Micro Presentation Spaces 19-6
Example Code: Creating and Associating a Presentation Space 19-7

Chapter 20. Graphic Primitives and Their Attributes 20-1
The Line Primitive 20-1
Attributes of Lines 20-3
The Marker Primitive 20-3
Attributes of Markers 20-3
The Image Primitive 20-4
Attributes of Images 20-4
Example Code 20-5
The Area Primitive 20-10
Alternate Mode 20-11
Winding Mode 20-11
Attributes of Areas 20-12
The Arc Primitives 20-12
The Current Arc Parameters 20-13
Drawing a Full Arc 20-14
Drawing a Partial Arc 20-14
Drawing a 3-Point Arc 20-16
The Fillet 20-16
The Spline 20-17
Attributes of Arc Primitives 20-18
Setting a Group of Attributes for a Primitive Type 20-18
Paths 20-18
Defining a Path 20-18

Chapter 21. The Color and Mix Attributes . 21-1
Color Tables 21-1
The Default Logical Color Table 21-2
Defining a Logical Color Table 21-3
The Color Attribute 21-5
The Mix Attribute 21-6
Drawing Color Graphics on Monochrome Devices 21-9
Drawing Color Area-Fill Patterns on Monochrome Devices 21-10

Chapter 22. Graphics Segments 22-1
The Drawing Mode 22-1
Creating a Graphics Segment 22-2
The Elements of a Segment 22-3
The Element Pointer 22-3
Adding Elements to a New Segment . . . 22-3
Element Types 22-4
Labels 22-4
Editing a Segment 22-4
Inserting or Replacing Data in a Graphics Segment 22-4
Locating Elements 22-5
Deleting a Part or the Whole of a Graphics Segment 22-5
Copying a Segment 22-5
The Attributes of a Segment 22-6
The Chained Attribute 22-6
The Fast-Chaining Attribute 22-7
The Dynamic Attribute 22-8
The Detectability Attribute 22-8
The Visibility Attribute 22-8
Changing the Attributes of a Segment . . 22-9

Drawing a Picture 22-9
Drawing Dynamic Segments 22-10
The Drawing Controls 22-10
Attribute Modes 22-11
Reusing the Presentation Space 22-11
The GpiSavePS (GSSPS) Call 22-12
The GpiRestorePS (GSRSPS) Call . . . 22-12
The GpiResetPS (GSRPS) Call 22-12
The GpiSetPS (GIPS) Call 22-12
Correlation 22-12
The Pick Aperture 22-13
Correlating on Retained Graphics 22-13
Correlating on Nonretained Graphics . . 22-14
Example Code: Creating Segments 22-15

Chapter 23. Graphics Text 23-1
The Presentation Manager-Supplied Fonts 23-1
Creating Fonts 23-1
Public and Private Fonts 23-2
Selecting a Font 23-2
Defining a Logical Font 23-2
Selecting the Current Font 23-5
Reassociating the Presentation Space . . . 23-5
Deleting Logical Fonts and Unloading Physical Fonts 23-6
The Code Page 23-6
Proportional Spacing and Kerning 23-7
The Character-String Primitive 23-8
Character Modes 23-8
Attributes of Character Strings 23-9
Formatting Text 23-13
Creating Marker Sets and Pattern Sets . . . 23-15
Using Bit Maps as Area-Fill Patterns . . 23-16
Example Code: Defining a Logical Font . 23-16

Chapter 24. Bit Maps in Graphics Applications 24-1
When to Use Bit Maps 24-1
Creating and Drawing Bit Maps 24-2
Creating Bit Maps Using the Icon Editor 24-3
Saving Bit Maps 24-4
Deleting Bit Maps 24-4
Making Bit Maps Available to Other Processes 24-4
The GpiBitBlt (GIBB) Call 24-5
Changing the Size of the Bit Map 24-5
Specifying a Mix Value 24-5
Converting between Monochrome and Color Data 24-6
The GpiWCBitBlt (GIWBB) Call 24-7
Example Code: Saving and Redisplaying Pictures in Bit Maps 24-7

Chapter 25. Transformation 25-1
Coordinate Spaces 25-1
World-Coordinate Space 25-1
Model Space 25-2
Presentation-Page Space 25-4
Device Space 25-8
The Windowing-System Transformation . 25-9
Coordinate Spaces Summary 25-10
Transforming Bit-Map Data 25-10
Specifying Transformations 25-11
Translating a Graphics Object 25-11
Scaling a Graphics Object 25-12
Rotating a Graphics Object 25-13
Transformation Matrixes 25-14
Coding Transformations 25-15
Accumulating Transformations 25-16
Concatenating Transformations 25-17
The Device-Transformation Matrix . . . 25-19
Shearing a Graphics Object 25-20
Example Code: Changing the Segment Transformation 25-20

Chapter 26. Clipping 26-1
How Clipping is Implemented 26-1
The Clipping Path 26-1
The Viewing Window 26-2
The Graphics Field 26-3
Regions 26-4
Example Code: Boundary Determination . . 26-9

Chapter 27. Saving and Restoring Graphics . 27-1
Metafiles 27-1
Creating a Metafile 27-2
Saving a Metafile 27-3
Loading a Metafile 27-3
Playing a Metafile into a Graphics Presentation Space 27-3
Making a Metafile Available to Other Applications at the Same Workstation . 27-5
Editing Metafiles 27-5
Copying Metafiles 27-5
Deleting Metafiles 27-5
Restrictions on Metafile Contents 27-5
PIF Files 27-7
The Convert Picture File Utility 27-7
Converting from Picture Interchange Format to Metafile Format 27-7

Part 5. Presentation Manager Applications — Data Transfer Between Running Applications

Chapter 28. User-Driven Data Exchange — The Clipboard 28-1
Application View of the Clipboard 28-1
Clipboard Owner 28-3

The CLIPBRD Sample PM Program 28-3
Program Description 28-3
Clipboard Function Call Summary 28-6

Chapter 29. Application-Driven Dynamic Data Exchange 29-1
DDE is Message-Based 29-2
Terminology and Basic Rules 29-3
Uses for Dynamic Data Exchange 29-3
Inside DDE: the Workings of the Protocol . 29-4
DDE Protocol Definition 29-5
Conventions Used in This Chapter 29-5
DDE Message Set 29-5
DDE Implementation Rules 29-5
Initiation of a Conversation Between Two Applications 29-6
One-Time Data Transfer Between Two Applications 29-6
Permanent Data Link between Two Presentation Manager Applications . . . 29-7
Execution of Commands in a Remote Application 29-8
Termination of a Conversation between Two Applications 29-9
Synchronization Rules 29-9
Shared Memory Object Rules 29-10
Syntax of the EXECUTE Command String 29-10
The System Topic 29-10

Part 6. Alphanumeric Text Applications – User Input/Output

Chapter 30. Alphanumeric Video Output . . . 30-1
Alphanumeric Support for Presentation Manager Applications 30-1
Creating the Vio Presentation Space . . . 30-1
Associating the Window with the Presentation Space 30-2
Setting the Cell Size 30-2
Creating Logical Fonts 30-2
Defaults for Text-Windowed Applications . . 30-3
Alphanumeric Support for All Application Types 30-3
Directly Accessing the Logical Video Buffer 30-3
Performing String Updates to the Logical Video Buffer 30-4
Determining the Adapter Type 30-5
Enabling ANSI Support 30-5
Setting Video Code Pages 30-5
Requesting a Temporary Foreground Session 30-6
Manipulating the Cursor 30-6
Scrolling the Screen Image 30-7
Setting the Display Mode for Text-Windowed and Full-Screen Applications 30-7
Video Output Support Specific to Full-Screen Applications 30-9
Setting the State of the Adapter 30-9
Loading Fonts 30-9
Using Your Own Graphics Interfaces . . 30-10
Registering Another Video Subsystem . 30-12
Sharing the Vio Handle with Other Processes and Threads 30-13

Chapter 31. Keyboard and Mouse Input for Text-Windowed and Full-Screen Applications 31-1
Keyboard Input 31-1
Reading Keyboard Input 31-1
Accessing a Secondary Logical Keyboard 31-3
Managing the Keyboard Device 31-3
Registering Another Keyboard Subsystem 31-4
Mouse Input 31-5
Opening the Mouse Event Queue 31-5
Setting the State of the Event Queue . . . 31-5
Reading Data from the Event Queue . . . 31-6
Manipulating the Pointer 31-7
Registering Another Mouse Subsystem . . 31-8

Part 7. Control Program Functions

Chapter 32. Memory Management 32-1
Memory Management Design Considerations 32-1
Dynamic Data Segments 32-1
Creating and Accessing Private Segments 32-1
Creating and Accessing Shared Segments . 32-3
Accessing A Data Segment as a Code Segment 32-4
Creating and Accessing Discardable Segments 32-4
Suballocating Pieces of a Segment 32-4
Changing the Size of Allocated Memory . 32-5
Freeing Memory 32-7
Obtaining the Size of Allocated Memory . 32-7
Creating Resource Segments for Presentation Manager Applications 32-7
Resource Design Considerations 32-7
Accessing a Read-Only Resource 32-8
Freeing a Resource 32-8

Chapter 33. File and Disk Management . . . 33-1
Using OS/2 File Systems 33-1
Installing OS/2 File Systems 33-2
Recognizing DOS and OS/2 File Objects . 33-3
Using the High Performance File System 33-4
Managing OS/2 File Systems 33-5
Attaching and Detaching File Systems . . 33-5
Getting Information About a Device and an FSD 33-6
Communicating With a File System 33-6
Preparing File Systems For System Shutdown 33-6
Naming OS/2 File Objects 33-6
Creating Long File Names 33-7
Using Long File Names 33-8
Processing Meta Characters 33-9
Naming Character Devices 33-10
Managing Names of File Objects 33-11
Creating File Objects 33-11
Creating a File 33-11
Creating a Subdirectory 33-13
Managing File Objects 33-13
Managing Files 33-13
Managing File Handles 33-14
Managing Directories 33-15
Managing Logical Drives 33-16
Defining Information Associated with File Objects 33-16
Naming EAs 33-17
Indicating EA Value Data Types 33-18
Using Standard Extended Attributes . . 33-19
Managing Information Associated with File Objects 33-23
Managing Extended Attributes 33-23
Manipulating File Objects with EAs 33-26
Searching for File Objects 33-28
Reading and Writing Data 33-30

Chapter 34. Program Execution and Control 34-1
Threads and Processes 34-1
Starting Another Thread of Execution 34-1
Suspending Threads within a Process . . . 34-2
Ending the Current Thread or Process . . 34-3
Multithreading Considerations for PM Applications 34-4
Object Ownership 34-7
Owner Windows 34-7
Scheduling 34-7
Shared Data 34-8
Private Data 34-8
Window Handling 34-8
Shutting Down Multithreaded Multiwindowed Applications 34-8
Starting Another Process 34-9
Setting Priorities for Threads and Child Processes 34-10
Starting Another Session 34-11
Starting a Child Session 34-12
Notifying a Process When a Child Session Ends 34-12
Requesting the Foreground 34-12
Controlling Child Sessions 34-12
Session Function Call Restrictions 34-13
Timer Services 34-13
Setting a Synchronous Timer 34-14
Setting an Asynchronous One-Shot Timer 34-14
Setting an Asynchronous Recurring Timer 34-14
Dynamic Linking 34-15
Linking to a Dynamic Link Module at Run Time 34-16
Debugging Aid to Program Authors 34-17

Chapter 35. Communications Between Threads and Processes 35-1
Creating and Accessing Semaphores 35-1
System Semaphores 35-2
RAM Semaphores 35-2
Serializing the Use of Resources 35-3
Waiting for the Occurrence of an Event . 35-4
Setting Flags to Respond to Time-Critical Events 35-6
Communicating with Pipes and Queues . . . 35-7
Passing Data in an Unnamed Pipe 35-7
Passing Addresses to Data with a Queue . . . 35-8
Ownership and Management of a Queue . 35-8
Client Access to a Queue 35-9
Managing Queue Data Storage 35-10
Two-Way Communication with Named Pipes 35-11
Ownership and Management of the Named Pipe 35-11
Accessing the Named Pipe by a Client . 35-13
Writing to a Named Pipe 35-14
Reading a Named Pipe 35-14
Synchronizing Pipe Dialogs 35-15
Transaction Processing on Duplex Message Pipes 35-15
Getting Named Pipe Information 35-15
Coordinating Access to a Local Named Pipe 35-16

Part 8. Printing and Device Support

Chapter 36. Printing 36-1
The Spooler 36-1
Benefits of Using the Spooler 36-1
Overview of the Spooling Process 36-2

Spool-File Formats 36-3
The Role of the Presentation Driver 36-3
The Role of the Print Manager 36-5
The Role of the Spooler Queue Processor 36-5
Printing Alphanumeric and Other Raw Data 36-6
Using the Device Function Calls 36-6
Using the Dos Function Calls 36-7
Printing Graphic Data 36-7
Printing Graphics Text in Text Mode . . . 36-8
Printing Graphics Using Print Picture . . . 36-9
Printing a Copy of the Screen 36-10
Using a Separate Thread for Printing . . . 36-12
Handling Print Errors 36-12
Stopping the Creation of a Print Job . . 36-12
Using the Spooler Directly 36-12
Printing Without the Spooler 36-13
What if the Spooler is Not Selected? . . 36-13
Example Code: Printing Using a Separate Thread 36-14
Example Code: Printing Using Device Calls 36-16

Chapter 37. Creating a Printer Device Context 37-1
The DevOpenDC (WDOPEN) Call 37-1
The Elements of the DEVOPENDATA Structure 37-2
Selecting the Output Device 37-3
Selecting the Queue Processor 37-4
Getting Form Selections and Job Properties . 37-4
Querying the Presentation Driver 37-5
Passing Parameters to the PMPRINT Queue Processor 37-7
Examples Using the ARE and FIT Parameters 37-8
Example Code: Creating a Printer Device Context 37-10

Chapter 38. Device Support 38-1
Device Drivers 38-1
Application I/O to Devices 38-1
The File System Interface 38-1
The Subsystem Interface 38-1
The IOCtl Interface 38-1
Character Device Monitors 38-1
IOPL Code Segments 38-2
Presentation Drivers 38-3

Part 9. Product Development Considerations

Chapter 39. Supporting National Languages 39-1
Resources and National Language Translation 39-1
Selecting a Language Version of an Application 39-1
Translating Messages in Full-Screen Applications 39-2
Code Page Support 39-2
Code Page Management 39-2
Code Page Preparation 39-3
Code Page Operation 39-5
Code-Page Switching 39-7
Code Page Supported Devices 39-10
Application Design Considerations . . . 39-10
Special Considerations and Limitations . 39-10
Code Page API Summary 39-12

Chapter 40. Migrating Applications to the Presentation Manager 40-1
The Decision to Migrate 40-1
Migrating PC-DOS Applications 40-1
Migrating a Microsoft Windows Application 40-1
Migrating Microsoft Windows Graphics Applications 40-2
Migrating OS/2 Full-Screen Applications to Presentation Manager Applications 40-2
Migrating OS/2 1.0 Applications to Text-Windowed Applications 40-3
Migrating OS/2 1.0 Applications to Presentation Manager Applications 40-3
Restructuring the Application 40-4
Migrating the Vio Calls to the Presentation Manager 40-4

Chapter 41. Non-Program Products That You Can Develop Using OS/2 41-1
Function Libraries 41-1
Window Libraries 41-1
Resource Libraries 41-1
Graphics Libraries 41-1
Loadable Font Libraries 41-1
Installation Programs 41-1

Appendixes

Appendix A. API Error Processing A-1
Control Program API Processing A-1
Presentation Manager API Processing A-1

Appendix B. GPI Calls B-1

Appendix C. Graphics Attributes C-1

Appendix D. Graphics Orders D-1
The Graphics-Orders Header File (PMORD.H) D-1
Decoding Graphics Orders D-2
Naming Conventions D-2

Appendix E. Dos, Kbd, Mou, and Vio Calls by Application Type E-1

Appendix F. PC-DOS Interrupts and Related OS/2 Function Calls F-1

Appendix G. System Limits G-1
Recovery Operations G-1
Memory Resource Limits G-1
Session Limits G-2
Presentation Manager Limits G-2
Hardware Limits G-2
Limits for Various Operating System Resources G-3

Appendix H. Information Presentation Facility Tags H-1
.* (Comment) H-2
:artlink. (Artlink) H-3
:artwork. (Artwork) H-5
:caution. (Caution) H-8
:cgraphic. (Character graphic) H-9
:color. (Color) H-10
:dd. (Definition description) H-11
:ddhd. (Definition description head) H-12
:dl. (Definition list) H-13
:dt. (Definition term) H-15
:dthd. (Definition term head) H-16
:fig. (Figure) H-17
:figcap. (Figure caption) H-18
:fn. (Footnote) H-20
:h1. through :h6. (Headings) H-21
:hide. (Hide) H-24
:hp. (Highlight phrases) H-26
:i1. / :i2. (Index) H-28
:icmd. (Index command) H-30
.im (Imbed) H-31
:isyn. (Index synonym) H-32
:li. (List item) H-34
:lines. (Lines) H-35
:link. (Link) H-37
:lm. (Left margin) H-40
:lp. (List part) H-42
:note. (Note) H-43
:nt. (Note) H-44
:ol. (Ordered list) H-45
:p. (Paragraph) H-46
:parml. (Parameter list) H-47
:pd. (Parameter description) H-49
:pt. (Parameter term) H-50
:rm. (Right margin) H-51
:sl. (Simple list) H-53
:ul. (Unordered list) H-55
:userdoc. (User document) H-57
:warning. (Warning) H-58
:xmp. (Example) H-59

Appendix I. Information Presentation Facility Symbols I-1

Appendix J. OS/2 1.2 Conformance with Common User Access J-1

Glossary X-1

Index X-13

Part 1. Introduction

Chapter 1. Introduction to the IBM Operating System/2 Application Programming Interface

This chapter describes the environments that the IBM Operating System/2 (OS/2) provides for running application programs. It describes the classes of application program supported in each environment, and introduces the subsets of the OS/2 application programming interface (API) that you can use to write applications of each class.

One of the API subsets, called the Presentation Manager API, is based on the IBM Systems Application Architecture (SAA) Common Programming Interface – a software interface for the design and development of applications that can be run in the three major IBM computing environments:

- Personal Computer (Personal System/2 (PS/2) or IBM Personal Computer AT under OS/2)
- System 370 (TSO/E under MVS/XA, and CMS under VM)
- AS/400.

The Presentation Manager API also implements SAA's Common User Access which you can use to attain consistency in appearance and behavior of the user interface of your applications.

OS/2-Mode and DOS-Mode

OS/2 provides two different modes in which applications are run. The modes are *OS/2 mode* and *DOS mode*. In OS/2 mode, several applications can run concurrently. In addition, parts of applications written in programming languages that support reentrancy can also run concurrently with each other and with applications. This is called *multitasking*. Each running application and its data are protected from being accidentally changed by any other concurrently running application. DOS mode is described in "DOS-Mode Applications" on page 1-4.

In OS/2 mode there are three levels of multitasking: *sessions*, *processes*, and *threads*. If required, you can use the levels to divide the functions of your application into separate logical areas.

A session is the top-level object in the multitasking hierarchy. A session usually consists of a process, a logical display screen, and logical input devices. OS/2 can run several sessions concurrently, and the user can switch between them. The user can interact with only one session at any one time. The session with which the user interacts is called the foreground session. All the other sessions continue to run but receive no input from the user.

A process is an executing application plus the resources it uses.

A thread is an executable unit contained in a process. When a process starts running, it contains only one thread, at the entry point of the process. In C/2 and Macro Assembler/2, the thread can start one or more additional threads running. Threads run concurrently, in a similar way to processes. However, threads share the resources of the process that created them. A thread is like a subroutine that can be called by the main routine of your application, but can run concurrently with the main routine.

COBOL/2 and FORTRAN/2 applications do not have access to the OS/2 thread functions.

See Chapter 34, "Program Execution and Control" for a full description of multitasking.

Concurrently running processes and threads need to communicate with one another, to coordinate their order of running and their access to shared resources. This is called *interprocess communication*, and OS/2 provides several methods. See Chapter 35, "Communications Between Threads and Processes," for a full description.

OS/2-Mode and DOS-Mode Application Types

OS/2 can run two types of application in OS/2-mode sessions:

- Applications that run in windows
- Full-screen applications.

OS/2-Mode Applications that Run in Windows

You can write an OS/2-mode application using a subset of the OS/2 input/output (I/O) API that allows the application to run in a rectangular subdivision of the display screen called a *window*. A windowed application can share the screen with other windowed applications, as shown in Figure 1-1.

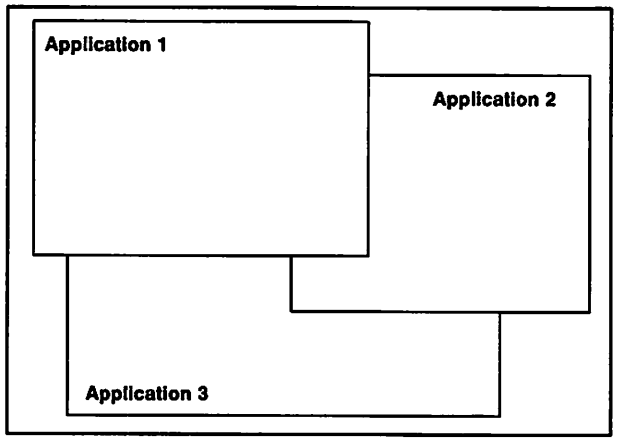


Figure 1-1. Windowed Applications Can Share the Screen

The user can start windowed applications from the *User Shell*. The User Shell is a component of OS/2 that uses a graphics-based windowed user interface to present a visual means of managing the applications and files installed and running under OS/2. Through the User Shell, using a mouse or keyboard, the user has a high level of interaction with OS/2 and can run several applications simultaneously. (The User Shell effectively replaces the old PC-DOS-style command line, and protects the user from the complexities of the system and its commands. However, a command line is also available for users who prefer that type of interface.)

Each windowed application runs in its own session under OS/2, and can appear in a window on the display screen concurrently with other windowed applications and the User Shell, which also run in their own sessions. When the user selects an application with which to interact, OS/2 puts that application's session in the foreground. Usually, the main window of a windowed application can be resized to fill the entire screen.

Windowed applications can be subdivided into three specific classes of application:

- Presentation Manager applications
- Dialog Manager applications
- Text-windowed applications.

Presentation Manager Applications

A Presentation Manager (PM) application explicitly creates one or more windows to present function to the user. *A PM application should interact in a compatible way with OS/2, and other PM applications (including the User Shell).* In particular, *a PM application should cater to the needs of the user.* These requirements directly affect how you design and write your applications, and mean that *you must put code in your applications to fulfil them.* One obvious requirement is, if you want your application to perform well under OS/2, the design of your application must reflect your awareness that the application is sharing the system with other applications. In particular, a PM application must do the following:

- Use OS/2-format messages to communicate with OS/2, and with other PM applications, and notify them of events that could affect them.
- Process message input in a particular way, and in a specific time limit, and get back to the user. For example, a C/2 or Macro Assembler/2 application that has lengthy processing to do should use the multitasking facilities of OS/2 to do it in a separate thread, and return to the user in the main thread. This allows the user either to use other functions in the application, or to use another application, while the processing completes.

 The PM API is available to COBOL/2 and FORTRAN/2 applications, and is suitable for highly interactive applications. However, multitasking is not available to COBOL/2 or FORTRAN/2 applications. If applications in these languages have a potentially time-consuming operation, for example, lengthy processing, or waiting for a facility beyond their direct control, they are unable to start a separate thread for the operation and get back to the user. The user cannot interrupt them, or

interact with other applications, until the processing is complete. For this reason, the PM API may not be suitable for such applications.

In addition to the above requirements, it is recommended that you do the following:

- A PM application should use the OS/2 API to produce a graphics-based windowed user interface that conforms in appearance and usability with the SAA Common User Access definition, and with the User Shell. You should provide a user interface that makes the functions of your application *visible and easily accessible* to the user.

 The User Shell is a good example of such an interface; screen items representing the underlying function can be easily accessed, either by using a mouse to move a pointer around the screen and pressing a single mouse button, or by using the keyboard. This is unlike command-based applications where the user types in text instructions to use the function. Conforming with the Common User Access definition produces applications that, though they vary in function, have interfaces familiar to the user. This reduces the time the user spends in becoming familiar, and therefore productive, with your application.
- A PM application should supply information about itself to OS/2. Some of this information can be displayed to the user by the User Shell and can make it easier for the user to start and run your application. For example, when a PM application starts running, it supplies a user-friendly application name to the Task List, which the User Shell displays in a list of running applications.
- OS/2 supplies a storage area (not visible to the user) known as the *clipboard*. A pair of running PM applications can use the clipboard as an intermediate holding area for the user-controlled transfer of data. A PM application that provides this facility to the user must contain code that lets the user transfer the data in or out of the clipboard, and should use standard data formats.
- Text and similar resources displayed by your application should be kept separate from your application code. As a consequence, you can maintain much of the user interface of your application without altering the code. If your application is intended for an international market, this makes it easier to produce various national language versions of your application.
- You can make your application device-independent, because the OS/2 API is not device-specific. The OS/2 API accesses device drivers for a wide variety of output devices. For example, your application can find out the characteristics of the device on which it is running. Your application can then respond appropriately, for example, by adjusting the size of its window or windows, and the amount of data displayed in them, to suit the size of the display screen and the available text fonts.

A PM application that conforms to the above requirements will run efficiently and responsibly under OS/2 with other applications and the User Shell, and will serve the user with a highly interactive and responsive interface.

A PM application has to be specified as such to OS/2 when the application is installed.

See "What is in the OS/2 1.2 Application Programming Interface?" on page 1-6 for an introduction to the API subsets used by PM applications.

Dialog Manager Applications

The Dialog Manager provides a dialog tag language (DTL) for specifying panels, a DTL compiler to compile the panels, and a programming interface that calls Dialog Manager services to display and manage the panels. The Dialog Manager services are actually a high-level interface to the Presentation Manager functions, but do not allow your program the same degree of control of the user interface as using the Presentation Manager API directly. You can, however, use several of the calls in the PM API in a DM application. The Dialog Manager is described in the *OS/2 1.2 Dialog Manager Programming Guide and Reference* manual.

Text-Windowed Applications

A text-windowed application does *not* explicitly create a window or windows in which to run on the screen. Instead, OS/2 supplies a text-windowed application with a default window in which to run. A text-windowed application uses an API subset that does not allow it to communicate with OS/2 and other applications, but is compatible with them. A text-windowed application can, therefore, share the display screen with other applications that run in windows. The API subset is primarily for displaying IBM 3270 display-style fixed-size alpha-

numeric text. It could be used, for example, by a compiler or editor application so it can share the screen with other applications.

If the calls in a text-windowed application are all contained in an API subset known as the *family API*, you can use a utility called *BIND* to convert the executable OS/2 application into the equivalent of an executable DOS application, called a family application. A family application can be transferred to any member of the IBM PC family that runs under PC-DOS, and then run as a PC-DOS application. However, the family API does not take advantage of many of the OS/2-mode facilities. Family API calls are identified as such in the *Control Program Programming Reference* manual.

A text-windowed application has to be specified as such to OS/2 when the application is installed, unless it has been linked with the WINDOWCOMPAT keyword. (Also, see the WINABLE utility on 2-14.)

See "What is in the OS/2 1.2 Application Programming Interface?" on page 1-6 for an introduction to the API subsets used by text-windowed applications.

Full-Screen Applications

You can write an application using a subset of the OS/2 API that does not allow the application to share the display device with any other concurrently running application.

The user can start up to 12 full-screen applications from the User Shell. When running, each full-screen application runs in its own session separately from the User Shell. When the user chooses to interact with a full-screen application, that application's session becomes the foreground session. The application takes over the *entire* display screen until the user ends the interaction. When a full-screen OS/2 mode application is controlling the screen, other OS/2 mode applications continue to run concurrently, though the user cannot see them or interact with them. This is shown in Figure 1-2 on page 1-5.

A full-screen application can be written using the family API, but, again, in most cases it is better to write a family API application to be a text-windowed application, so that it can share the screen with other applications.

See "What is in the OS/2 1.2 Application Programming Interface?" on page 1-6 for an introduction to the API subsets used by full-screen applications.

DOS-Mode Applications

OS/2 emulates PC-DOS, and calls it *DOS mode*. By selecting the DOS mode icon, the user can enter DOS mode from the User Shell. As in PC-DOS, the user is given a full screen and a command line. From here the user can run a single PC DOS application, and (with a few possible restrictions) it will run exactly as it did under PC-DOS. An application running in DOS mode takes over the *entire* display screen, and runs in a single DOS session.

A DOS-mode application can be written using either DOS interrupts or the family API.

If the user does not need to run family or PC-DOS applications, OS/2 can be installed without DOS mode. A DOS-mode application has to be specified as such to OS/2 when the application is installed.

See "What is in the OS/2 1.2 Application Programming Interface?" on page 1-6 for an introduction to the API subset used by DOS-mode applications.

How the OS/2 Application Classes Run Concurrently

To summarize, the classes of application that OS/2 supports are as follows:

- OS/2-mode applications:
 - PM applications
 - Text-windowed applications
 - Dialog Manager applications
 - Full-screen applications.
- DOS-mode applications.

Figure 1-2 on page 1-5 shows what the user sees when running three windowed applications A, B, and C (these can be PM or text-windowed applications) a full-screen application D, and a DOS-mode application E. The figure also shows the sessions that relate to the applications, and which is the foreground session.

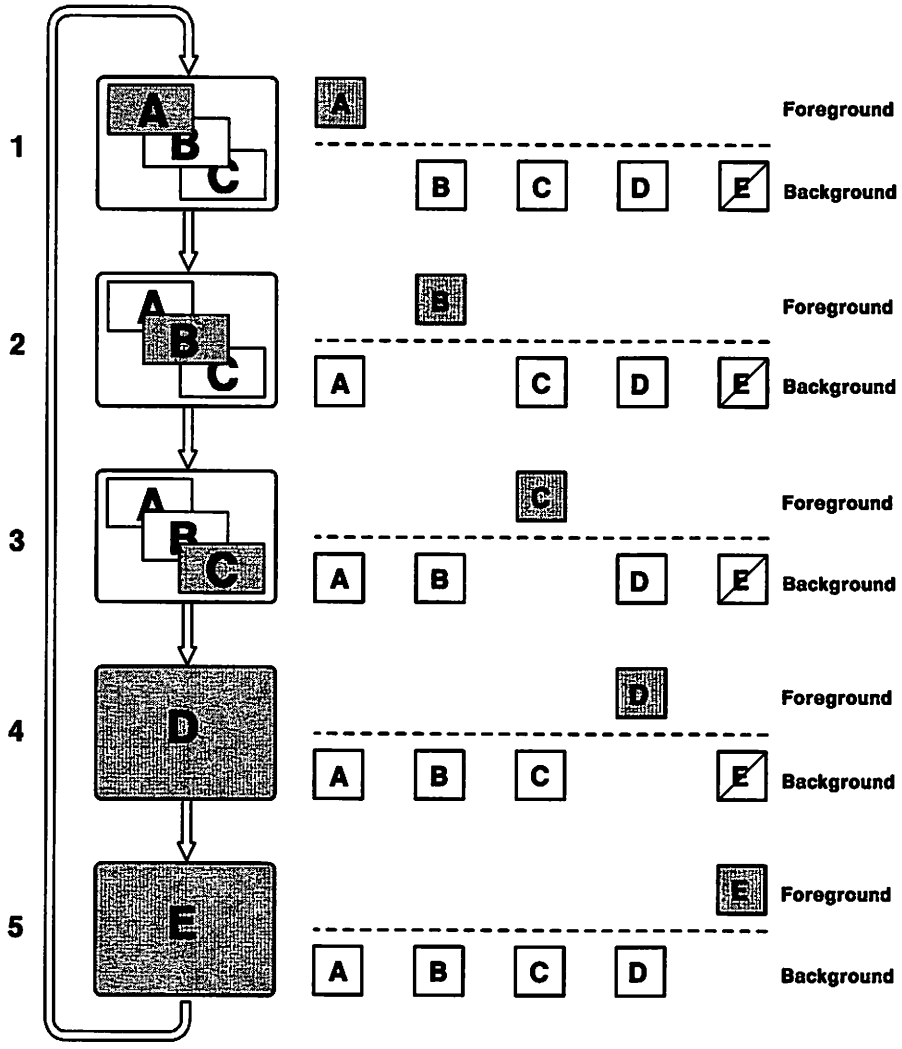


Figure 1-2. OS/2-Mode Applications and a DOS-Mode Application Running Concurrently. A, B, and C are PM or text-windowed applications. D is a full-screen application. E is a DOS-mode application.

The user can use a combination of keys to switch to each running application in turn. The user can first switch through all the applications running in windows, and then switch through all the applications running in other sessions. The visual effect of this is similar to switching between the channels on a television set, where one channel can display several television programs at the same time.

The diagram shows the switching sequence as follows:

1. The first screen shows the three windowed applications A, B, and C running concurrently. Although application D cannot be seen, it is running. The user is interacting with the topmost window, which belongs to application A. Application A is the foreground session. However, when the single DOS-mode session containing a family or DOS application is not the foreground session, it is always suspended. This prevents it from interfering with the OS/2-mode applications.
2. The user is interacting with the topmost window, which belongs to application B. Application B is the foreground session. The session containing E is still suspended.
3. The user is interacting with the topmost window, which belongs to application C. Application C is the foreground session. The session containing E is still suspended.
4. The user is interacting with the full-screen application D. Application D is the foreground session. The session containing E is still suspended.

5. The user is interacting with E. The DOS session is no longer suspended, but is running in foreground. When a DOS session is running in foreground, the other OS/2-mode applications are still running, but receive no input from the user.

OS/2 Mode and How It Breaks the 640KB Barrier

In DOS mode, as in PC-DOS, less than 640KB of real addressable random access memory are available to an application. This barrier prevents the user from running a large application.

OS/2 mode uses the addressing capabilities of the Intel 80286 and 80386 chips to break the 640KB barrier. OS/2 mode supports up to 16MB of real addressable memory. Any application and data can, therefore, be up to almost 16MB.

OS/2 mode also supports the overcommitment of memory. When physical memory size is less than an application requires, OS/2 can swap segments of the application to the fixed disk, and bring them back as needed. This feature means that applications can actually be up to 128MB in size.

See Chapter 32, "Memory Management" for a full description of memory management.

How an Application Uses the OS/2 Application Programming Interface

Under OS/2, the code segments that relate to the OS/2 API calls are kept in OS/2-supplied libraries, and can be dynamically linked by your application at application load time. This is called *dynamic linking*.

Several applications can share the dynamic link libraries that contain the code relating to the OS/2 API. This results in smaller applications than under PC-DOS, where any code segments external to an application (such as the code relating to DOS functions, or code for your own called subroutines) were statically appended to the application's .EXE file at link-edit time. This could result in a large .EXE file.

Creating Your Own Dynamic Link Libraries

Under OS/2, you can create dynamic link libraries, containing your own code. Your applications can share the code in the libraries, as required.

If a subroutine is infrequently called, your application can delay dynamically linking its code segment until it is used by the application. For example, your application does not have to load an error-processing routine unless an error occurs during running. See the *Building Programs* manual for details of how to build dynamic link libraries, and see "Dynamic Linking" on page 34-15 for how to use them in an application.

What is in the OS/2 1.2 Application Programming Interface?

OS/2 function calls are coded in mixed case in C/2, and uppercase in COBOL/2, FORTRAN/2, and Macro Assembler/2.

In C/2 and Macro Assembler/2, calls begin with a three-letter prefix indicating the general API subset. The prefixes are:

Dev Device support

Dos Control Program

Gpi Graphics programming interface

Kbd Keyboard direct access

Mou Mouse direct access

Pic Picture support

Prf Profiles (initialization files)

Spl Spooler

Vio Alphanumeric text (video output)

Win Windows.

The prefix is followed by several words, or abbreviations of words, that describe the function performed by the call.

In COBOL/2 and FORTRAN/2, the calls begin with a one-letter prefix, followed by an abbreviation of the function performed by the call. Windows calls start with W. Graphics calls start with G.

The next four sections of this chapter give an indication within application class, of the functional areas addressed by the OS/2 Version 1.2 API, and which functional areas contain the family API. Exact details of restrictions are available in the call descriptions in the *Control Program Programming Reference*.

The API Calls Used By Windowed Applications

Presentation Manager Applications

Presentation Manager applications can share the screen, keyboard, mouse, and printers with other windowed applications and the User Shell. A call that tries to access a device *directly* is behaving in a manner incompatible with the sharing of the device by other applications and the User Shell. Such a call is, therefore, not allowed in a Presentation Manager application.

Presentation Manager applications can use subsets of the OS/2 API that support the following general functional areas:

Windows A subset that your application can use to create a windowed user interface. "Part 2. Presentation Manager Applications – Using the Windowed Programming Interface" describes how to use the windowing calls.

Graphics A subset for the creation, storage, and retrieval of graphics text and pictures that are displayed in windows. "Part 4. Presentation Manager Applications – Using the Graphics Programming Interface" describes how to use the graphics calls.

The windowing and graphics calls together largely support SAA. For the cross-system portability considerations of application design, see *Systems Application Architecture: Writing Applications, A Design Guide*. The SAA publications are listed in the preface to the *Programming Overview*.

Presentation Manager applications can also use the following calls:

Alphanumeric text A subset for the display of fixed-size alphanumeric text in windows that your application creates explicitly. "Part 6. Alphanumeric Text Applications – User Input/Output" describes the alphanumeric text calls that can be used in a Presentation Manager application.

Device support A subset for querying and accessing devices.

Spooling A subset for writing data direct to a spool file. "Using the Spooler Directly" on page 36-12 describes how to use these calls.

Profile calls A subset for querying or writing to user-specific and system-specific initialization files.

DOS All of the DOS calls except the device monitoring calls. The Dos calls are described in "Part 7. Control Program Functions."

In COBOL/2 and FORTRAN/2, the control program, keyboard and mouse direct access, and alphanumeric text calls are not supported.

Dialog Manager Applications

Dialog Manager applications use the Dialog Manager services to display panels in windows, and can use calls from the same function call areas as PM applications, but with restrictions in some areas. See the *OS/2 1.2 Dialog Manager Programming Guide and Reference* for details.

The API Calls Used By Text-Windowed Applications

Text-windowed applications can use subsets of the OS/2 API that support the following general functional areas:

Alphanumeric text A subset for the display of fixed-size alphanumeric text in text windows that OS/2 creates by default for your application. "Part 6. Alphanumeric Text Applications – User Input/Output" describes the alphanumeric text calls that can be used in a PM application.

Keyboard A text-window-compatible subset of calls that directly access the keyboard.

Mouse A text-window-compatible subset of calls that directly access the mouse. The keyboard and mouse subsets are described in Chapter 31, "Keyboard and Mouse Input for Text-Windowed and Full-Screen Applications."

DOS All of the Dos calls except the device monitoring calls. The DOS calls are described in "Part 7. Control Program Functions."

If a text-windowed application uses only calls from the above groups that are also in the family API subset, the application is portable to any member of the Personal Computer family that runs under PC-DOS. However, the family subset does not contain facilities for multitasking, interprocess communication, segment swapping, and dynamic linking.

A text-windowed application can also use the calls that have the following prefixes:

Device support A subset for querying and accessing devices.
Spooling A subset for spooling print output.

See "Part 6. Alphanumeric Text Applications – User Input/Output" for a description of user input/output calls for text-windowed applications.

The API Calls Used By Full-Screen Applications

Full-screen applications can use any of the keyboard, mouse, and Dos calls, and a full-screen-compatible subset of the alphanumeric calls.

If a full-screen application uses only calls from the above groups that are also in the family API subset, the application is portable to any member of the Personal Computer family that runs under PC-DOS. However, the family subset does not contain facilities for multitasking, interprocess communication, segment swapping, and dynamic linking.

Also, full-screen applications can use the calls that have the following prefixes:

Device support A subset for querying and accessing devices
Spooling A subset for spooling print output.

See "Part 6. Alphanumeric Text Applications – User Input/Output" for a description of user input/output for full-screen applications.

The API Calls Used By DOS-Mode Applications

DOS-mode applications can use any of the family subset of the calls in the OS/2 Version 1.2 API. However, the family subset does not contain facilities for multitasking, interprocess communication, segment swapping, and dynamic linking.

See "Part 6. Alphanumeric Text Applications – User Input/Output" for a description of the user input/output family API subset.

Language Support

OS/2 supports the following languages:

IBM Basic Compiler/2
IBM COBOL/2
IBM C/2
IBM FORTRAN/2
IBM Macro Assembler/2
IBM Pascal Compiler/2

IBM Operating System/2 does not provide the same level of support to all programming languages. You should be aware of the areas in which the languages are not fully compatible with OS/2.

You can use the Presentation Manager API only in C/2 Version 1.1, Macro Assembler/2 Version 1.0, COBOL/2, and FORTRAN/2. Because COBOL and FORTRAN do not support reentrancy, the support for the Presentation Manager API is limited in these languages.

You can use the alphanumeric, keyboard, mouse, and DOS APIs only in C/2 Version 1.1 and Macro Assembler/2 Version 1.0. Only C/2 Version 1.1 and Macro Assembler/2 Version 1.0 provide full support for multithread applications and dynamic link libraries.

You cannot use any of the OS/2 API from a Basic or Pascal program.

Two C/2 run-time libraries support creation of dynamic link libraries. A single-thread application that calls subroutines in dynamic link libraries can link to the LLIBDLL.LIB run-time library. A multithread application can link to the CDLLOBJS.LIB run-time library.

Earlier C/2 run-time libraries were not reentrant, meaning that two threads could not use them at the same time; a second thread could overwrite data in the first thread. The supported version of C/2 provides a reentrant version of the standard run-time library, LLIBCMT.LIB, which serves the needs of a multithread application. Alternatively, a multithread application can control access to C/2 run-time libraries by using a semaphore. Note, however, that not all C/2 run-time libraries are reentrant. The C/2 manual contains a list of all the libraries that are reentrant.

Subroutines you have created and put into dynamic link libraries should not contain static or global variables if you want the subroutines to be reentrant. If the subroutines contain such variables, the application must maintain separate static and global variables for each thread identifier that accesses the subroutine.

COBOL/2 Version 1.0 supports only one entry point in each dynamic link library. Refer to your language manual for additional information.

Applications written in all supported programming languages must use the Pascal calling convention to access the IBM Operating System/2 services. The supported compilers (and the assembler) automatically use the Pascal calling convention if the application includes the header files, OS2.H or OS2.INC, provided in the programming tools. If the application does not include the header files, the compilers default to the C/2 calling convention, which causes the application to fail. Including the header files is recommended, but you can set the Pascal compiler option or write your own function prototype, declaring each function as FAR Pascal.

386 Programming Considerations

When executing on a system with a 386 processor, OS/2 now preserves the 386 extended registers across all context switches. These switches include thread switch, process switch and preemption due to a hardware interrupt. The registers preserved include:

EAX

EBX

ESI

EDI

EBP

32-Bit

These additional 16-bit 386 segment registers are also preserved:

FS

GS

Note that this does not imply that segments of greater than 64KB are supported. It does imply that the high-order word of the following registers will be destroyed on a context switch:

ESP

EIP

Chapter 2. OS/2 1.2 Programming Tools

To install the Programming Tools, follow the instructions in the *OS/2 1.2 Programming Tools Installation* card. The tools consist of the following:

- Include and header files
- Sample programs
- Family Application programming utilities
 - BIND Utility
 - Import Librarian
 - Message Utilities.
- Presentation Manager programming utilities
 - Dialog Box Editor
 - Icon Editor
 - Font Editor
 - Resource Compiler
 - WINABLE.
- Dialog Manager
- Dialog Tag Language Compiler
- Help Manager Utilities
 - Information Presentation Facility language Compiler
 - GML to IPF Conversion Utility.

This chapter introduces many of these tools. However, for details about the Dialog Manager and the Dialog Tag Language Compiler, see the *OS/2 1.2 Dialog Manager Programming Guide and Reference.*

Include and Header Files

The Programming Tools contain source header files (file-name extension .H) for C programmers, and source include files for Macro Assembler/2, COBOL, and FORTRAN programmers. Macro Assembler/2 include files have a .INC extension, COBOL include files have a .CIN extension, and FORTRAN include files have a .FIN extension. These save you programming effort.

C/2 and Macro Assembler Files

A statement at the top of your source code file automatically calls a hierarchy of header or include files. The C/2 and Macro Assembler/2 include files let you tailor which groups of functions you want to include. You can select the files you want to include by placing statements in your source code **before** the include statement.

For C/2 the statements are as follows:

```
#define INCL_*
#include <os2.h>
```

For Macro Assembler/2 the statements are as follows:

```
INCL_* equ 1
include os2.inc
```

In both cases INCL_* represents a symbolic identifier where * represents an abbreviation that defines the include file or part of the include file where the call is declared. To save you from defining large numbers of symbols, most of the files have a single symbol allowing all the functions to be accessible. See the relevant *Bindings Reference* for details.

The following files are available at compile or assemble time:

Level 1 — OS/2 and Presentation Manager

OS2.H and OS2.INC
Includes:

OS2DEF(.H or .INC)

BSE(.H or .INC)

PM(.H or .INC)

Level 2 — OS/2

OS2DEF.H and OS2DEF.INC
Define common constants, types, error codes, and structures. They set up some useful macros.

BSE.H and BSE.INC
Allow you to set up defined symbols to include all the OS/2 API functions, including Dos, Kbd, Vio and Mou. Include the following files:

BSEDOS(.H or .INC)

BSESUB(.H or .INC)

BSEERR(.H or .INC)

Level 3 — OS/2

BSEDOS.H and BSEDOS.INC
Set up constants, structures, and function prototypes for the Dos API calls.

BSESUB.H and BSESUB.INC
Set up constants, structures and function prototypes for the Vio, Kbd, and Mou API calls.

BSEERR.H and BSEER.INC
Set up the error code constants for all the OS/2 API calls.

Level 2 — Presentation Manager

PM.H and PM.INC
Enable you to include Presentation Manager API functions selectively by defining symbols. Include the following files:

PMWIN(.H or .INC)

PMGPI(.H or .INC)

PMDEV(.H or .INC)

PMAVIO(.H or .INC)

PMSPL(.H or .INC)

PMPIC(.H or .INC)

PMORD(.H or .INC)

PMBITMAP(.H or .INC)

PMFONT(.H or .INC)

Level 3 — Presentation Manager

PMWIN.H and PMWIN.INC
Set up constants, types, structures, and prototypes for the windows, message manager, keyboard, mouse, and dialog manager API functions.

PMGPI.H and PMGPI.INC
Set up constants, types, structures, and prototypes for the graphics API calls.

PMDEV.H and PMDEV.INC
Set up constants, types, and function prototypes for the device API calls.

PMAVIO.H and PMAVIO.INC
Set up constants, types, and function prototypes for the Presentation Manager Vio API calls.

PMSPL.H and PMSPL.INC
Set up constants, types, and function prototypes for the spooler API calls.

PMPIC.H and PMPIC.INC
Set up constants, types, and prototypes for the picture API calls.

PMORD.H and PMORD.INC
Set up the structures and constants that can be used to build or interpret GOCA orders for the graphics API calls.

PMBITMAP.H and PMBITMAP.INC
Set up types for bit maps.

PMFONT.H and PMFONT.INC
Set up types for fonts.

COBOL/2 and FORTRAN/2 Include Files

The COBOL/2 and FORTRAN/2 constants are stored in the following files:

- For Presentation Manager, PM.CIN and PM.FIN
- For OS/2, OS.CIN, and OS.FIN.

These files are very large, therefore it is recommended that you copy the constants you require into your own include file. You can then include this file in your program, instead of the entire include file.

To call your include file automatically, add the following statement to your source code file:

For COBOL/2: `COPY 'MYPROG.CIN'`

For FORTRAN/2: `INCLUDE 'MYPROG.FIN'`

There is no support for the selective inclusion of PM include files, or parts of PM include files, as there is for C/2 and MASM/2. There is no equivalent to the OS.H file and the underlying structure of include files that are supplied for C/2 and MASM/2.

COBOL/2 and FORTRAN/2 Structures

Some PM functions use structures for input and output. You must define the structures in your program.

For COBOL/2, OS/2 supplies a set of COPY files (one for each structure) containing a structure of PICTURE descriptions. The first item in the structure is declared with a level number of 10. Your application provides the declaration for the level 01 item, and uses the COPY statement to include the OS/2-supplied structure. See "COBOL Coding Techniques" in the *COBOL/2 Bindings Reference* for more information.

For FORTRAN/2, OS/2 supplies a set of INCLUDE files (one for each structure) where each structure is overlaid on a character string of equivalent size to the structure as a whole. If you need to use more than one instance of a structure in your application, you can use a copy technique, or repeat the definition of the structure with a different name. See "FORTRAN Coding Techniques" in the *FORTRAN/2 Bindings Reference* for more information.

The following list shows the names of these files. In each case a COBOL file (.CIN) and a FORTAN file (.FIN) is supplied. The equivalent long names for the data types are shown for each file name.

File Name	Long Name
ACCEL	ACCEL
ACCELTAB	ACCELTABLE
ARCPARML	ACCELPARAMS
AREABUN	AREABUNDLE
BANDRECT	BANDRECT
BITINF	BITMAPINFO
BITINFH	BITMAPINFOHEADER
BTNCDATA	BTNCDATA
CHARBUN	CHARBUNDLE
CLASSINF	CLASSINFO
CRTSTRUC	CREATESTRUCT
CURINF	CURSORINFO
DEVSTRUC	DEVOPENSTRUC
DLGT	DLGTEMPLATE
DLGTITEM	DLGTITEM
ENTFDATA	ENTRYFDATA
FATTRS	FATTRS
FFDESCS	FFDESCS
FMETRS	FONTMETRICS
FRACDATA	FRAMECDATA
HCINFO	HCINFO
HPROGA	HPROGARRAY
IMAGEBUN	IMAGEBUNDLE
KERNPAIR	KERNINGPAIRS
LINEBUN	LINEBUNDLE
MARGST	MARGSTRUCT
MARKBUN	MARKERBUNDLE
MATRIXLF	MATRIXLF
MENUITEM	MENUITEM
MLECDATA	MLECTLDATA
MQINF	MQINFO
OVERFLOW	OVERFLOW
OWNERIT	OWNERITEM
PROGENTR	PROGRAMENTRY
PROGTYPE	PROGTYPE
PTRINF	POINTERINFO
QMSG	QMSG
QPDATA	SQPOPENDATA
RGB	RGB
RGNRECT	RGNRECT
SBCDATA	SBCDATA
SMH	SMHSTRUCT
SWBLOCK	SWBLOCK
SWCNTRL	SWCNTRL
SWENTRY	SWENTRY
SWP	SWP
TRACKINF	TRACKINFO
USERBTN	USERBUTTON
WNDPARMS	WNDPARMS
XYWINSIZ	XYWINSIZE

The PM Sample Programs in C/2

When installed, the samples that run under Presentation Manager appear in a group in the Desktop Manager window. The first time you start one of the samples, Desktop Manager calls a .CMD file which compiles the source code of the sample into its executable form, and then runs the program. A subsequent selection of a PM sample from Desktop Manager runs the executable program.

The "Hello" C Sample – HELLO1

This program introduces the fundamentals of PM programming. The program displays a window with a red background containing some text in white. It says **Hello** to the user, and gives the instructions for using the program:

Mouse button 1 changes the color of this window

F3 terminates the program

When the user presses mouse button 1 with the pointer in the window, the background color switches between red and blue. When the user presses F3, the program ends.

The code of HELLO1 is explained in Chapter 4, "The HELLO1 Sample Application."

The Standard Window C Sample – HELLO2

HELLO2 introduces the elements of an OS/2-supplied window class called a standard window. It uses a standard window to display its main window. The standard window has an action bar containing the single choice **Options**. The **Options** choice has a pull-down with three choices that each paint a different string in the window, and an **Exit** choice.

The code of HELLO2 is explained in Chapter 6, "The HELLO2 Sample Application."

The Dialog Box Introductory C Sample – DIALOG1

DIALOG1 shows how to associate a pop-up window with a standard window. The program uses a standard window to display its main window. The standard window has an action bar with the choice **Options**.

The Options Pull-Down

The **Options** pull-down contains the following choices:

Where

This displays a pop-up window containing text which asks **Where do you live?** and an entry field in which the user can type the name of a town or city. The pop-up window also contains **OK** and **Cancel** pushbuttons.

The pop-up window is implemented in the program using an OS/2-supplied window class called a dialog box. A pushbutton is an example of another type of OS/2-supplied window called a control. When the user selects **OK**, the name of the town or city appears in the window, at the end of a string of text **You live in**.

Exit

Selecting Exit quits the program. The F3 key can be used as an alternative method to exit the program. A message is displayed to confirm the request.

The code of DIALOG1 is explained in Chapter 9, "DIALOG1 Dialog Box Sample Application."

The Dialog Boxes and Controls C Sample – DIALOG2

DIALOG2 builds on the ideas already introduced. It uses a standard window to display its main window. From the standard window one of several dialog boxes can be displayed. The dialog boxes use several of the OS/2-supplied controls. User interaction with the controls determines what is displayed in the main window. DIALOG2 also demonstrates how to write an application whose user interface conforms to the Common User Access definition.

The standard window contains a system menu, size borders, title, and an action bar. The title contains the program name. The action bar contains the choices **Options**, and **Help**. The window contains the following words:

Entry Field 1 =
Entry Field 2 =
List Box Selection =
Combo Box Selection =

It also contains a further window called **Color sample** that initially is painted cyan (turquoise).

The Options Pull-Down

The **Options** pull-down contains the following choices:

Entry field example
Auto radio button example
Check box example
List box example
Combo box example
Modeless dialog example
Exit.

When one of these choices, other than **Exit**, is selected, a dialog box is displayed.

Entry field example
The entry field example dialog box lets the user enter two integers in the range 1 through 100. There are pushbuttons for **OK**, **Cancel**, and **Help**. When the user selects **OK**, the fields are validated. If an error is found, a message is displayed. When the message box is dismissed, the cursor is in the entry field in error.

If the fields are valid and **OK** is selected, the dialog disappears and the main window is updated with the values entered. For example:

Entry Field 1 = **42**
Entry Field 2 = **98**
List Box Selection =

The **Cancel** button key destroys the dialog and does not update the wording in the main window.

Auto radio button example
The auto radio button dialog contains three circular buttons (radio buttons) labeled **Red**, **Green**, and **Blue**, and a preview area that initially is the same color as the color sample in the main window. When a radio button is selected, the preview area is repainted immediately in the selected color. When the dialog is terminated by **OK**, the color sample is set to the preview color.

Check box example
The check box dialog is similar to the auto radio button dialog. However, in this case, one or more choices can be checked, and the preview area is always repainted immediately in the mixture of colors selected. When the dialog is terminated by **OK**, the color sample is set to the preview color.

List box example
The listbox dialog box displays a scrollable list of the names of the letters of the Greek alphabet. If the user selects **OK** to end the dialog, the selected name is written in the main window. For example:

List Box Selection = Omega

If the user selects **Cancel** to end the dialog, the wording in the window is not updated.

Combo box example
The combo box example displays a scrollable list of the names of the letter of the Greek alphabet, together with an entry field. The user may make a single selection, or alternatively enter a text string in the field provided. If the user selects **OK** to end the dialog, the selected name or string is written in the main window. If the user selects **Cancel** to end the dialog, the wording in the window is not updated.

Modeless dialog example
The modeless dialog example displays a list of text colors from which the user may make a single selection. If the user selects **Apply**, the selected color is saved and forces a repaint of the client window. If the user selects **Undo**, the current color is discarded and the client window is repainted to the previously selected color. If the user selects **Cancel**, the color of the text in the client window remains the same.

Exit
Selecting Exit quits the program. The F3 key can be used as an alternative method to exit the program. A message is displayed to confirm the request.

The Help Pull-Down

The **Help** pull-down contains four choices:

Extended help
Keys help
Help index
About.

Help is displayed for each of the first three choices when they are selected. Alternatively, the F1 function key can be used to display general program help.

When **About** is selected, the logo panel for the application is displayed.

The code of DIALOG2 is explained in Chapter 10, "The DIALOG2 Sample Application."

Note: In this sample help is not implemented as a function, it only shows what the pull-down should look like and how each selection should result in the display of a help window. The proper help function with an example of how to invoke the Help Manager is included in the TEMPLATE sample program, see "The Template C Sample – TEMPLATE" on page 2-11.

The Retained Graphics C Sample – GRAPHIC1

GRAPHIC1 covers the following tasks:

- Storing a picture
- Drawing a picture in a window
- Repainting a picture because of system windowing events
- Scrolling or zooming a picture
- Dragging a picked segment
- Printing window/screen contents using the spooler.

The program also demonstrates how a program is well-behaved, by using a lower priority asynchronous application thread to do all the drawing. When performing the above tasks, the program monitors its message queue at short intervals. Finally, the program demonstrates the tasks that are required on termination.

GRAPHIC1 displays a standard window containing a picture consisting of two boxes and several other smaller shapes. The mouse can be used to click inside one of the smaller shapes. The shape is drawn in outline and, if partially obscured, is redrawn on top. By moving the mouse, the shape can be dragged to anywhere in the picture. Releasing the mouse button redraws the shape in its normal color.

The displayed picture is scaled to fit the width of the window but is twice as tall. A vertical scroll bar is included in the standard window for vertical scrolling to be used to view all the picture.

The action bar contains the choices **Options**, and **Help**.

The Options Pull-Down

The **Options** pull-down contains four choices:

Zoom In
Zoom Out
Print
Exit.

Zoom In and **Zoom Out**
These choices can be used to increase or decrease the scale of the picture. There is a finite limit to how far the user can zoom in either direction. The relevant choice is disabled at each limit.

Print
This choice prints the picture currently displayed on the screen.

Exit
This choice has the same effect as in DIALOG2. It is described in Exit on page 2-5.

The Help Pull-Down

This pull-down has the same effect as in DIALOG2. It is described in The Help Pull-Down section of DIALOG2 on page 2-6.

The Non-Retained Graphics C Sample – GRAPHIC2

GRAPHIC2 is designed to display and optionally print a selected metafile. As in GRAPHIC1, the picture is scrollable. Drawing is performed by an asynchronous application thread, allowing the program to be well behaved.

A simple embedded picture is displayed in the window. However, the user can select an alternative picture from a selected metafile library. The action bar contains the choices **File**, **Options**, and **Help**.

The File Pull-Down

The **File** pull-down contains three choices:

Open
Print
Exit.

Open
This displays a dialog box containing a title bar, two list boxes, and three pushbuttons. The user can do the following:

- Select a metafile for display in the main window. While the picture is being updated, the user interface is still active, although no drawing or action may be obvious.
- Select an alternative directory, resulting in the first list box being updated with the metafile names from the new directory.
- Cancel the choice, without selecting a new file for display.

Print
This prints the displayed picture.

Exit
This has the same effect as in DIALOG2. It is described in Exit on page 2-5.

The Options Pull-Down

The **Options** pull-down contains one choice: **Best Fit**.

It sizes the picture so that the whole picture is displayed and its aspect ratio is preserved.

The Help Pull-Down

This pull-down has the same effect as in DIALOG2. It is described in The Help Pull-Down section of DIALOG2 on page 2-6.

The Clipboard C Sample — CLIPBRD

CLIPBRD displays a picture on which you can perform the clipboard operations: cut, copy, and paste. The picture is stored in memory as a bit map. The program illustrates the use of various clipboard, windowing, and graphics functions.

The mouse is used to define a rectangle within the picture, the contents of which you can cut or copy to the clipboard. You can also delete the contents of the rectangle or fill (paste) it from the clipboard.

The action bar contains the choices **Edit** and **Help**.

The Edit Pull-Down

The **Edit** pull-down contains the following choices:

Cut
Copy
Paste
Clear
Exit.

Initially all the choices on this pull-down, except **Exit**, are grayed. The other choices become selectable when you define a rectangular area within the picture. To do this, position the mouse pointer at the top-left corner of the area that you want to mark, then drag the mouse downwards and to the right to define the complete area. However, the **Paste** choice remains grayed unless there is some bit-map data already in the clipboard.

The pull-down choices have the following effects:

Cut
Stores in the clipboard the part of the picture defined by the rectangle, erases that part of the picture, and removes the defining rectangle from the window.

Copy
Copies into the clipboard the part of the picture defined by the rectangle, and removes the defining rectangle from the window.

Both **Cut** and **Copy** destroy any bit-map data already in the clipboard because it can hold only one item of a particular data type at a time.

Paste
Copies into the defining rectangle whatever bit-map data is held in the clipboard. The data is stretched or compressed to conform to the size of the rectangle. The data is retained in the clipboard, so it can be pasted many times.

Clear
Erases the part of the picture defined by the rectangle.

Exit
This choice has the same effect as in DIALOG2. It is described in Exit on page 2-5.

The Help Pull-Down

This pull-down has the same effect as in DIALOG2. It is described in The Help Pull-Down section of DIALOG2 on page 2-6.

The code of the CLIPBRD sample is explained in Chapter 28, "User-Driven Data Exchange – The Clipboard."

The Image C Sample – IMAGE

This sample program displays an image using a file of image data as the picture source. The file specified for selection is accessed, and the picture displayed in one of two ways. In **browse** mode, the maximum picture area is displayed in a window. Areas of the image off screen can be scrolled using the horizontal/vertical scroll bars.

Because of the nature of image display, restrictions are implemented to limit sizing of the window to the picture boundary. Normal sizing, using the sizing border, is restricted to the picture boundary.

You can choose to **view** the entire picture, which uses a different method that allows stretching and compression.

The action bar contains the choices **File**, **Options**, and **Help**.

The File Pull-Down

The **File** pull-down contains the following choices:

Open
This displays two dialog boxes. The first contains a list of file names on the current directory that conform to the image data file name specification. The second contains a list of alternative directories. The user can do the following:

- Select a file for display within the list box. On selection of the **Open** button, the entire image is scaled to fit the main window before it is displayed. This demonstrates the **view** method.
- Select an alternative directory, resulting in the list box being updated with image data file names from the new directory.
- Cancel the choice, without selecting a new file for display.

Exit
This choice has the same effect as in DIALOG2. It is described in Exit on page 2-5.

The Options Pull-Down

The **Options** pull-down contains the following choices:

Browse
Displays the unscaled image in the window. Parts of the picture outside the window can be scrolled into view using the scroll bars. This choice can be selected only when the application is in view mode.

View
The image is redrawn, scaled to fit the window. This choice can be selected only when the application is in browse mode.

Colors
A dialog box is displayed containing radio buttons for selection of the foreground and background colors in the main image window. On selection of the **OK** pushbutton, the main image is redrawn using the specified colors.

The user can quit this selection by using the **Cancel** pushbutton, which resumes normal program operation without altering the image colors.

Save Position
In browse mode, the user scrolls the main image to bring into view any section of the image which can be off screen. The coordinates of any given position can be stored for later recall by selecting this choice. This choice is active only in browse mode.

Restore Position
Selection of this choice resets the position of the main image to that saved using the **Save Position** option. This choice is active only in browse mode.

The Help Pull-Down

This pull-down has the same effect as in DIALOG2. It is described in The Help Pull-Down section of DIALOG2 on page 2-6.

The Bit Map C Sample — BMAP

When it is started, the BMAP sample program displays a window that contains a line of text. The text gives the name of the program. The action bar contains the choices **Options** and **Help**. The program illustrates the creation and redrawing of a bit map. Two methods are provided for redrawing externally-created bit maps that are held in a resource file:

1. using WinDrawBitmap (WIDBMP)
2. using GpiBitBlt (GIBB).

The program also demonstrates the use of graphics calls to draw graphics primitives.

The Options Pull-Down

This displays a pull-down containing the following choices:

WinDrawBitmap
This choice is always selectable, and it is checked while it is the current method used to refresh the window. A predefined bit map is loaded from a resource file and is displayed using the WinDrawBitmap (WIDBMP) call. This bit map is stretched and compressed to conform to the window when it is resized.

GpiBitBlt
This choice is always selectable, and it is checked while it is the current method used to refresh the window. A predefined bit map is loaded from a resource file and is displayed using the GpiBitBlt (GIBB) call. This bit map is stretched and compressed to conform to the window when it is resized.

Draw Primitives
This choice is always selectable, and it is checked while it is the current method used to refresh the window. A simple non-retained picture is drawn in a presentation space using graphics drawing orders.

Selection of this choice makes the **Save Picture** choice selectable.

Save Picture
This choice is selectable only when the **Draw Primitives** choice is the current method used to refresh the window. The coordinates of a rectangle surrounding the complete picture are calculated. If the picture is clipped by the window, a message box is displayed and you are asked if you still want to save the picture. If you continue, the part of the picture that is visible in the window is copied to a bit map. Selection of this choice makes the **ReDraw Picture** choice selectable.

ReDraw Picture
This choice is selectable if the **Save Picture** choice is selected during the current session of the program. It is checked while it is the current method used to refresh the window. The bit map that was created last time the **Save Picture** choice was selected is drawn. It is not stretched to fit the window. The window may need to be sized to see the bit-map version of the simple drawing. The part of the window not filled by the bit map is painted cyan.

Exit
This choice has the same effect as in DIALOG2. It is described in Exit on page 2-5.

The Help Pull-Down

This pull-down has the same effect as in DIALOG2. It is described in The Help Pull-Down section of DIALOG2 on page 2-6.

The Advanced Vio C Sample — AVIOSAMP

This sample program provides the facilities of a simple text editor, allowing the user to create new files and edit existing files. There is a limit to the size of file that can be accessed. The action bar contains the choices **File**, **Options**, and **Help**.

The File Pull-Down

This pull-down contains the following choices:

Open
Presents a dialog box containing a list box of file names in the current directory. The user can:

- Select a file for use from the listbox. Selecting the **Select** button displays the file text in the main window.
- Select an alternative directory. The listbox is updated with file names from the new directory.
- Cancel the choice, without selecting a new file.

New
Enables the user to enter text and use the editing functions.

Save
Saves the file under its current name.

Save as
Presents a dialog box which contains a list box and an entry field. The listbox contains a list of drives and directories from which to choose and the entry field allows the user to enter a file name. The file is saved to the chosen drive and directory disk using the entered file name.

Print
Prints file text on the system printer.

Exit
This choice has the same effect as in DIALOG2. It is described in Exit on page 2-5.

The Options Pull-Down

This pull-down contains a single choice: **Color**. It displays a dialog box containing radio buttons for the selection of foreground and background colors. When the user selects **OK**, text in the window is redrawn using these colors.

The user can select **Cancel** to quit this choice. Normal program operation continues without altering the colors.

The Help Pull-Down

This pull-down has the same effect as in DIALOG2. It is described in The Help Pull-Down section of DIALOG2 on page 2-6.

Special Keys

The following functions are available in AVIOSAMP using the ordinary keys:

- TAB
- NEWLINE
- DELETE CHAR
- BACKSPACE
- INSERT ON/OFF.

Key combinations (which you can display using **Extended Help**) are used to perform the following actions:

- SPLIT LINE
- JOIN LINE
- ERASE LINE
- DELETE LINE
- ADD LINE
- ERASE TO END OF LINE
- UNDERLINE ON/OFF
- REVERSE VIDEO ON/OFF.

The cursor can be positioned either by using the cursor up/down/left/right keys, or by clicking the left mouse button. The cursor is set at the position indicated by the mouse pointer within the screen window. Horizontal cursor keys must be used to see the effect of the underline and reverse video options. These are treated as separate formatting facilities, and do not occur with normal character input. This formatting is only available for screen output, and is ignored when printing text. There is a status line at the bottom of the window which displays the state of any items that can be switched on or off.

The Typing C Sample — TYPETEXT

TYPETEXT is designed to show how character input is handled and echoed in a window.

The program displays a window and allows the user to type in one line of text, up to 80 characters long. The system beeps when the user tries to enter more than 80 characters.

A flashing cursor indicates where the next character will be displayed and this is moved on the line as characters are entered.

The backspace key deletes a character. The Enter or Newline keys delete a whole line and place the cursor back at the beginning of the line. No other special keys are handled.

The Font Test C Sample — FONTTEST

FONTTEST shows the application programmer how to find out about the fonts available and how to use fonts to display character strings.

The action bar contains the choices **Fonts** and **Help**.

The Fonts Pull-Down

This pull-down contains the following choices:

Select fonts
This choice displays a dialog box listing all the Presentation Manager-supplied fonts. When the name of a font is selected it is displayed in the center of the screen, and a character string is displayed at the bottom of the screen in the selected font. The selected font becomes the current font. When the dialog box is subsequently displayed, the current font is shown.

Modify a font
This choice displays a dialog box which allows the user to set various properties of a font. When **OK** is selected, the text **User Modified Font** is displayed. The line of text at the bottom of the screen is displayed in the closest possible font that the system can create to match the values the user has entered. In some cases it can appear that the font has not changed properly according to the values entered, but it is the best the system can do. The values set become the current values. When the dialog box is subsequently displayed, the current values are shown.

Exit
This choice has the same effect as in DIALOG2. It is described in Exit on page 2-5.

Both dialog boxes contain a **Help** pushbutton that displays a message showing where help for the dialog box should go.

The Help Pull-Down

This pull-down has the same effect as in DIALOG2. It is described in The Help Pull-Down section of DIALOG2 on page 2-6.

The Template C Sample — TEMPLATE

TEMPLATE is a skeleton program that can be the base for any application. You can start with this outline and fill in the code unique to your application.

TEMPLATE displays a standard window with an action bar containing the choices **Example** and **Help**.

The Example Pull-Down

This pull-down contains the following choices:

Dialog Example
This choice displays a simple dialog box from which a name can be selected for display in the main window. The dialog box contains a **Help** pushbutton that displays contextual help using the Help Manager.

Exit
This choice has the same effect as in DIALOG2. It is described in Exit on page 2-5.

The Help Pull-Down

This pull-down contains the following choices:

How to get help
Extended help
Keys help
Help index
About.

Special help panels are displayed for each of the first four choices when they are selected. Alternatively, the F1 function key can be used to display contextual help.

When **About** is selected, the logo panel for the application is displayed.

Programming Notes About TEMPLATE

The sample shows how an application can be split over more than one source file so that more than one code segment is used. The files are TEMPINIT, TEMPRES, and TEMPNRES. The sample gives an indication of what could go in the various types of files.

The dialog box is defined in a separate file so it can be used with the Dialog Box Editor. Note that the separate include file containing just the constants used with the dialog is required so that the Dialog Box Editor can be used.

Help in this sample is displayed by invoking the Help Manager.

The Hello World Macro Assembler/2 Sample — HELLOM

HELLOM is a simple program that demonstrates how Presentation Manager calls are used in Assembler. It is unlikely that anyone would write a complete application in Assembler. However, some calls are shown in case it is necessary to program in Assembler.

The program creates a standard window and displays the text **Hello world** in the lower-left corner of the window.

The PM Sample Programs in COBOL and FORTRAN

COBOL and FORTRAN versions of the DIALOG1 and DIALOG2 C-sample programs are provided. The COBOL versions are named DLG1COB and DLG2COB, and the FORTRAN versions are named DLG1FOR and DLG2FOR.

The COBOL programs are described in detail in Chapter 11, "The DLG1 Nonreentrant Sample Application" and Chapter 12, "The DLG2 Nonreentrant Sample Application."

Control Program Sample Programs

Following is a list and description of the sample programs demonstrating programming techniques using IBM Operating System/2. Code is provided in both C and Macro Assembler (.C and .ASM filename extensions).

CRERR
This sample program demonstrates the handling of a critical error by generating a pop-up window containing an error message.

D_INIT
This sample program demonstrates how to initialize a dynamic link library.

DLNK1
This sample program demonstrates how to create a dynamic link library file that can be pre-loaded by the system loader.

DLNK2
This sample program demonstrates how to create a dynamic link library file that can be loaded when called by the application.

DLNK3
This sample program demonstrates how to create a dynamic link library file that can be explicitly loaded, called, and freed by the application.

DYNLINK
This sample program demonstrates three ways of accessing a dynamic link library. In the first, the dynamic link library is pre-loaded by the system loader. In the second, the dynamic link library is loaded when called by the application. In the third, the dynamic link library is explicitly loaded, called, and freed by the application.

MEMORY
This sample program demonstrates how to allocate a memory segment, sub-allocate within the segment, expand the allocated segment, de-allocate a block of memory, and de-allocate a shared segment.

MOUSE
This sample program demonstrates how to receive input from the mouse pointer and write it to the screen.

VIOSAMP
This sample program demonstrates how to receive keyboard input and write it to the screen.

WT
This sample program demonstrates how to retrieve the date from the system in two country-specific formats. The program retrieves a message from a message file using the MKMSGF utility and displays the date in the appropriate format on the screen.

The following sample programs are written in Macro Assembler only:

MONITOR.ASM
This sample program demonstrates how to structure a device monitor, how a device monitor filters data passed to it from a device driver, and how the data is returned to the device driver.

PROCESS.ASM, PROG1.ASM, PROG2.ASM
These sample programs demonstrate how to execute multiple processes, how to allocate shared memory, how to execute child processes asynchronously, and how queues (PROG1) and pipes (PROG2) are used to pass data among processes.

THREAD.ASM
This program demonstrates how to use semaphores to synchronize two threads of execution.

DEMODD.ASM
This sample program demonstrates how to structure a device driver, including the strategy routine and the timer handler. The program demonstrates how to send data to a monitor. The data received from the monitor is placed in the queue managed by the timer handler.

IOPL2.ASM
This sample program demonstrates how to call a subroutine with special I/O privilege.

PROTMODE.ASM
This sample program demonstrates how to execute a family application in the OS/2 environment.

REALMODE.ASM
This sample program demonstrates how to execute a family application in the PC-DOS environment.

Control Program Message Utilities

OS/2 provides an easy method for separating messages from your application.

Make Message File (MKMSGF)

MKMSGF creates an indexed .MSG file from a source message text file. The indexed file is used by your application.

Message Bind (MSGBIND)

Only some of the messages need to be included in the .EXE file. MSGBIND includes selected messages from the .MSG file to an .EXE module. MSGBIND uses the .MSG file and a script file (naming the .EXE modules and listing the specific messages to be bound to each) to bind a message segment to the application.

Presentation Manager Programming Utilities

Dialog Box Editor
An interactive *what-you-see-is-what-you-get* (WYSIWYG) editor that you use to create and modify dialog boxes and the controls and informative text that they contain. It is described in "The Dialog Box Editor" on page 7-21.

Icon Editor
An interactive editor that you use to edit an enlarged version of an icon, pointer, or bit map in an editing window, while a true-scale version is displayed in a viewing window. It is described in "The Icon Editor" on page 8-1.

Font Editor
An interactive editor that you use to edit an enlarged version of a letter, character or symbol in an editing window, while a true-scale version is displayed in a viewing window. It is described in "The Font Editor" on page 8-8.

Resource Compiler
A compiler that uses an input file called a resource script file. The resource script file defines text-based application resources such as menus and text strings, and supplies the file names of non-text-based application resources such as icons, pointers, and bit maps. The resource compiler converts the resources into a binary-format executable file. It is described in "Writing a Resource Script File" on page 8-5, and in *Building Programs*.

WINABLE
The WINABLE utility converts many OS/2 full-screen applications so that they can run in a Presentation Manager window. See "Migrating OS/2 Full-Screen Applications to Presentation Manager Applications" on page 40-2 and *Building Programs* for details.

Help Manager Utilities

Information Presentation Facility language Compiler (IPFC)
A compiler that takes as input a file containing panels defined using Information Presentation Facility language (IPF) and converts the panels into an executable format. In Presentation Manager it is mainly used for converting the text of help panels.

GML to ITL Conversion Utility
A utility which converts text, written in dialog tag language, to a form suitable for input to the IPFC.

For more information on the Help Manager and its utilities, see "Part 3. Presentation Manager Applications – Using OS/2 Information Presentation Facility."

Part 2. Presentation Manager Applications – Using the Windowed Programming Interface

Chapter 3. Windows – Concepts and Fundamentals

This chapter describes Presentation Manager (PM) windows in the following terms:

1. A means to share, subdivide, and organize the screen layout.
2. A place to receive user input.
3. A place for applications to display data.

Sharing and Subdividing the Screen

A window is a rectangular subdivision of the screen. Every window belongs to a generic type called a *window class*, OS/2 supplies several window classes that have a standard appearance, and you can also register your own window classes. A PM application can create several windows of any class, each with its own data, and arrange the windows on the screen like papers on a desktop. The user can run several PM applications concurrently. The windows of each application share the screen with the windows of the other applications. Windows can overlap, with overlying windows obscuring parts of underlying, overlapped windows.

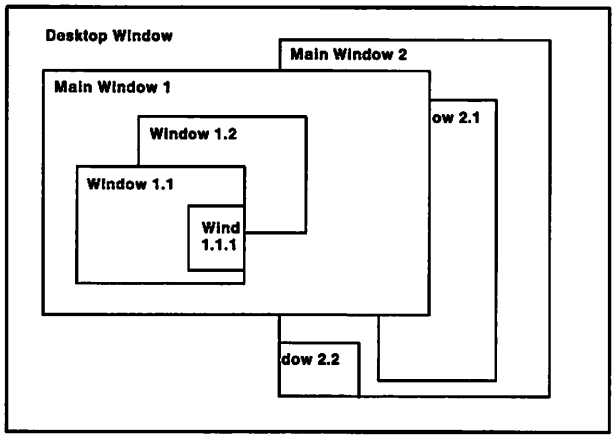


Figure 3-1. Several Windows on a Screen

Window Hierarchies

The windows on the screen exist in the system as a hierarchy of programming objects. At the top of the hierarchy is the *desktop window*, the display screen. Figure 3-1 represents the desktop window containing the windows of several applications. Figure 3-2 shows the corresponding logical hierarchy.

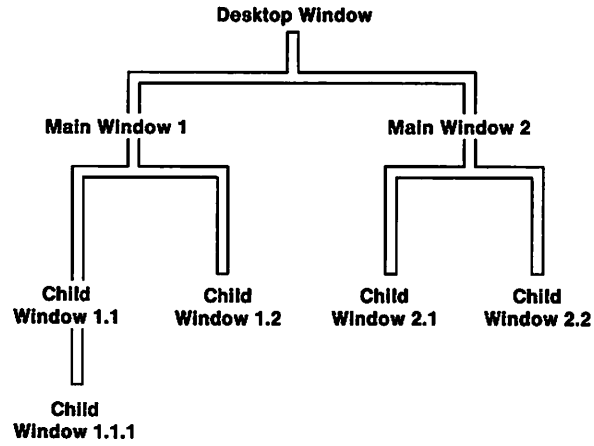


Figure 3-2. A Typical Window Hierarchy

The windows at the level immediately below the desktop are termed *main* or *top level* windows.

Window Relationships

The hierarchy defines various kinds of relationship between windows.

Parent-Child

In the logical hierarchy, the main window of each application is an immediate subordinate of the desktop window. The desktop is the parent, and each main window is a child, or subordinate, of the desktop.

In an application, you can extend the window hierarchy to organize data and function on the screen. You can separate different types of data into separate windows. This helps the user distinguish the data, and improves understanding of the function relating to each type of data. Subordinate to the main window of each application, multiple windows can exist at several levels in the hierarchy. The parent-child relationship occurs down the hierarchy. Because multiple levels are allowed, a child window can, at the same time, be a parent of other windows. Therefore, at each level in the hierarchy, the windows that are logically subordinate to an immediate superior are the child windows of that immediate superior (their parent).

You use child windows to subdivide the screen. Although a child window is beneath its parent in the logical hierarchy, it overlays its parent on the display screen. You can have windows within windows, so the space within a window is occupied by one or more additional windows that are dependent on the containing window.

A child window is subordinate to its parent in several ways. For example, when a window is moved, its child windows also move. The user can see only the portion of a child window that lies inside the boundary of its parent window. This effect is called *clipping*. When a parent window is removed from the screen, so are all its child windows.

Sibling Windows

Child windows that belong to the same parent are called sibling windows. The sibling relationship can occur at any particular level in the hierarchy, but applies only to the child windows belonging to *one* parent window. In the sample hierarchy in Figures 3-1 and 3-2, the two main windows are siblings, because they are both child windows of the desktop. At the next hierarchical level, windows 1.1 and 1.2 are siblings, because they are child windows of main window 1, and windows 2.1 and 2.2 are siblings, because they are child windows of main window 2.

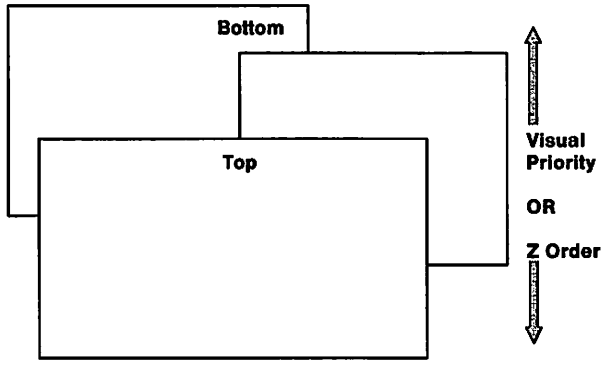


Figure 3-3. Window Viewing Order

Window Viewing Order: Each set of sibling windows has a *viewing order*, which defines the order in which they appear on the screen. The sibling viewing order is called the *z order*. (If x is the width of the screen, and y the height, then z is the depth of the screen.) The z order combines with the parent-child relationship to determine where each window appears in the "set of papers" on the "desktop". Sibling windows always overlay their parent on the screen, but the application can set or change their relative viewing order.

Window Coordinates

Window measurements are in a pel-based system called *window coordinates*. The x coordinate runs horizontally across the window from left to right. The y coordinate runs vertically up the window from bottom to top. The number space for window coordinates is from −32 768 to +32 767 pels, that is, it is 65 535 pels in size. The *origin* of a window is at its lower-left corner. Each window has a *size*, in both the x and y directions, relative to its origin. Windows can be any size within the window coordinate space.

You position a window by specifying the position of its origin relative to the origin of its parent. For example, main window 1, which has the desktop as its parent, is positioned relative to the lower-left corner of the desktop (the screen). Window 1.1, a child window of main window 1, is positioned relative to the lower-left corner of main window 1, and so on, through the hierarchy.

A window can be positioned anywhere relative to its parent. You can position the origin of a window outside its parent. However, only the part of a window which is inside its parent is visible, because the child window is clipped to the parent.

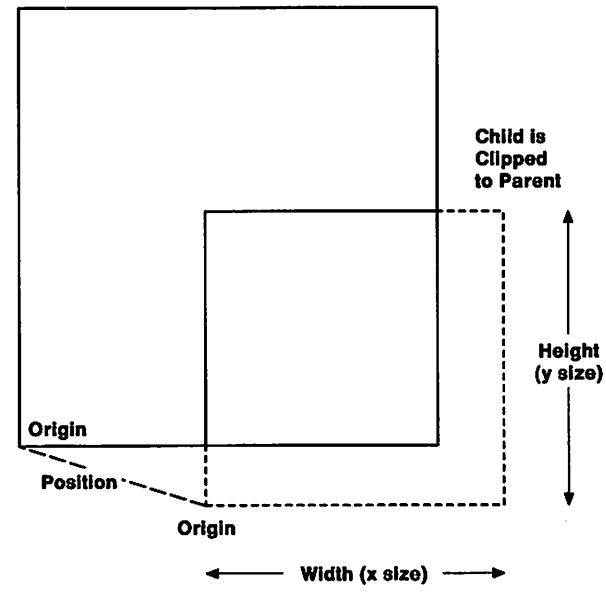


Figure 3-4. Window Positioning and Clipping

Input to Windows

The user directs input to a window from the mouse and the keyboard. PM applications generally use a window to display text and graphics on the screen, and to process input from the system keyboard, mouse, and timer. A window provides the same input and output capabilities as a virtual graphics terminal, without requiring direct control of the hardware.

Messages

User input can come from the mouse and the keyboard. Input is directed to the application by OS/2 in the form of *messages*. A message has four parameters:

- The *window handle* of the window to which the message is addressed. The handle is a unique number, held in a 32-bit word, that is returned to the application when it creates the window. The handle identifies the window, and distinguishes it from other windows on the screen.

 You should hold the information relating to a window in data structures based on this handle. OS/2 holds various items of system information such as the window size, position, and place in the hierarchy, and bases them on this handle.

- The *message type*. This is a 16-bit word containing a number. The number is defined in OS/2 as a symbolic name, consisting of a two-character prefix and a string of characters that indicates the purpose of the message. In C/2, FORTRAN/2, and MASM/2, the prefix and the string are separated by an underscore. In COBOL they are separated by a hyphen.

 The symbolic name indicates the source of the message. For example, when the user presses mouse button 1, OS/2 posts a WM_BUTTON1DOWN message.

- First *message parameter* (a 32-bit double-word).

- Second *message parameter* (a 32-bit double-word).

The message parameters contain data values that convey more information to the application. The parameters are of a generic data type that can contain a variety of data types such as long and short integers, Booleans, characters, pointers, and handles. The 32-bit message parameters are actually compound data types, (listed for each message in the *Control Program Programming Reference* manual) that can accommodate a variety of combinations of simpler data types, such as short (2-byte) and long (4-byte) integers, characters, pointers, and handles.

In C/2 and Macro Assembler/2, to help you extract the simpler data types from the compound data types, OS/2 provides a range of macros that decompose the message parameters:

PVOIDFROMMP
Extracts a pointer type.

HWNDFROMMP
Extracts a window handle.

CHARnFROMMP
Extracts the first, second, third, or fourth byte or character, depending on n, a number in the range 1 through 4.

SHORTnFROMMP
Extracts the first or second two bytes, depending on n, a number that is 1 or 2.

LONGFROMMP
Extracts all four bytes.

OS/2 also provides macros that you can use to compose messages.

For COBOL/2 and FORTRAN/2, there are utility calls for decomposing and composing messages. See "Constructing and Decomposing PM 4-byte Integers" on page 11-15 for details.

For full details of the macros or utilities, see the bindings reference manual for the language you are using.

Keyboard Messages

Keyboard input consists of key press and release actions. Keyboard input goes to one window at a time. The window receiving keyboard input is the *input focus window*. When a window is gaining the input focus, OS/2 sends a WM_SETFOCUS message with its second parameter set to TRUE. A window of an application that echoes alphanumeric keyboard input on the screen usually displays an OS/2-defined text cursor, showing where the next character appears when a key is pressed. When a window is losing the input focus, OS/2 sends a WM_SETFOCUS message with its second parameter set to FALSE. Applications that display graphics text can use the WinCreateCursor (WMCCUR), WinShowCursor (WISCUR),

WinDestroyCursor (WMDCUR), and WinQueryCursorInfo (WMQCUR) calls to control the cursor. Text-windowed applications in C/2 or Macro Assembler/2 can use VioSetCurPos, VioSetCurType, VioGetCurPos, and VioGetCurType.

For the keyboard, WM_CHAR messages are generated when keys are pressed and when they are released. Alphanumeric keys generate an ASCII character key code in the message. Function keys (F1 – F12, and so on) generate a virtual key code in the message, which is one of the set of VK_* values defined by OS/2. The complete list is in an appendix in the *Programming Reference: Volume 2* manual. (You can also define your own keys in the range VK_USERFIRST through VK_USERLAST.) The shift state of the keyboard is also returned, including standard shift, Ctrl shift, and Alt shift states. Typematic keys generate multiple key-down messages. OS/2 combines them into a single message, so that the input queue does not fill up with identical messages, and so that messages are not produced faster than the application can process them.

Normally, when the user presses a keyboard key, OS/2 generates a WM_CHAR message. However, in an file called a resource file you can specify instead that a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message with a particular *command value* is generated instead. A key that is defined in the resource file in this way is called an *accelerator key*. OS/2 tests the keyboard message to check if it is in the list of accelerator keys. If so, it translates the WM_CHAR message into the message type specified in the resource file. The original WM_CHAR message no longer exists, and the new message is directed to the window.

See "Controlling the State of the Keyboard" on page 13-18.

Mouse Messages

The mouse moves a pointer around the screen. The pointer is a small shape that appears on top of all other data on the screen and shows the mouse position on the screen.

The pointer is, by default, a small arrow, but applications can change the shape of the pointer on a window-by-window basis. Using the WinSetPointer (WMSPTR) call, the pointer can take up a different shape within each window. A range of system-provided pointers, that have identifiers with an SPTR_ prefix, are available to your application. The WinQuerySysPointer (WMQSPT) returns the handle of the specified SPTR_pointer, which you use as a parameter to WinSetPointer (WMSPTR). A commonly used system pointer is the SPTR_WAIT pointer, which is an hourglass, that an application usually uses to indicate that the application is performing some processing that makes mouse input momentarily unavailable. You can also create your own pointers, as described in "The Icon Editor" on page 8-1.

Mouse input goes to the window beneath the pointer. The precise position used for the pointer is called the *hot spot*. The hot spot is a single pel in the pointer shape. For example, in the default arrow, it is at the tip of the point. The input includes both mouse movement and mouse button press and release (click).

For the mouse, two types of input messages can occur. The first type is the WM_MOUSEMOVE message. This is generated when the mouse moves, and contains the position of the mouse in window coordinates. Again, OS/2 combines these messages into a single message. The second type of message is generated when the user presses or releases the mouse buttons. There are separate messages for each of the mouse buttons and for each of the mouse button actions: press down, release, and double-click. Double-click occurs when the user presses the mouse button twice in quick succession; the actual maximum time interval for this is user-specified in the Control Panel. Mouse messages are as follows:

WM_MOUSEMOVE
Mouse movement

WM_BUTTONnDBLCLK
Button n double-click, where n is the mouse button number

WM_BUTTONnDOWN
Button n down

WM_BUTTONnUP
Button n up.

The Active Window

The active window is the topmost main window with which the user is interacting, and usually has the input focus or contains a child window that has the input focus. This ensures that the user can see the window that accepts input. The active window is usually indicated by a highlighted title or border.

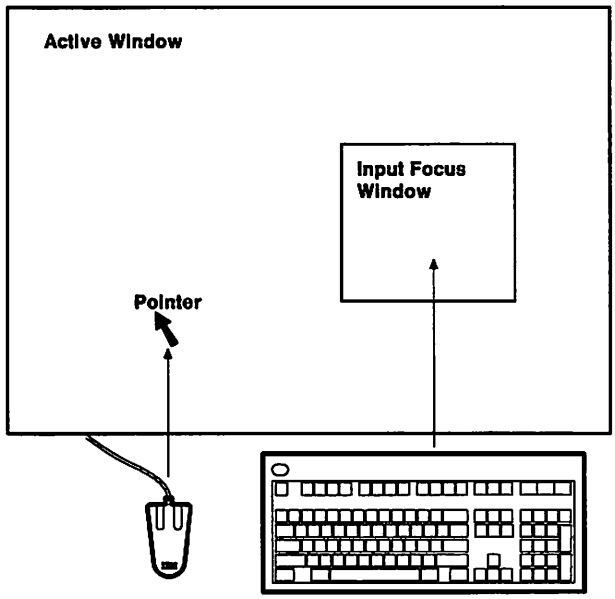


Figure 3-5. User Input to a Window

Normally, a main window is activated by the user clicking with the mouse in another main window or a child window of another main window, or by using the keyboard to switch to a new application and to another main window. The user can change the input focus in the same way. However, the application has more control over the input focus. For example, in a window containing several text entry fields, the tab keys can cause the input focus to move from one input field to the next, in an order defined by your application. When a main window is activated, it receives a WM_ACTIVATE message with its first parameter set to TRUE. When it is deactivated, it receives a WM_ACTIVATE message with its first parameter set to FALSE.

Messages Occur as a Result of Events

Messages are a fundamental part of OS/2. PM applications use messages to communicate with each other and OS/2. Communication with the system ensures concurrent running and sharing of devices. The function of a PM application (other than the primary function that is the reason for the existence of your application) is to process messages. A message notifies the receiving application that an *event* has occurred that could affect the receiver. OS/2 directs each message to a particular window, identified by the window handle in the message. Sources of events that cause a message to be sent to an application are:

The user
: As already mentioned, mouse or keyboard input to an application window causes OS/2 to direct messages to the window. The part of the application that controls the behavior and appearance of the window can respond to the input.

OS/2
: In managing the application windows on the screen OS/2 sends messages to the windows, usually indirectly as a result of user interaction. Messages bind the system together, and allow it to work in a well-ordered and uniform way. For example, where several application windows overlap, and the user terminates an application so that its window disappears, OS/2 directs messages to the underlying application windows so that they can repaint themselves.

The application itself
: An event can occur in the application to which another part of the same application should respond. For example, when the contents of its window no longer accurately reflects to the user the status of the application. In this case, the application sends *itself* a message telling itself to repaint the contents of the window. Messages can be used to access window-related functions. For example, the message *MM_INSERTITEM* is a menu message that inserts an item in a menu. The message is sent to the menu window in which the item is to appear. Part of the message data contains the actual definition of the menu item.

Another application
: Communication with other applications, through OS/2, ensures cooperative use of the system. With reentrant applications it can even be used to exchange data. For example, an arithmetic application that has finished a lengthy calculation can inform a business graphics application that it has some data for it to use.

The symbolic identifiers of OS/2-defined message types equate to an OS/2-internal range of values.

An application can define its own messages outside this range. See "Defining Your Own Messages" on page 13-3 for details.

Structure of PM Applications Written in a Reentrant Language

PM applications and OS/2 form an event-based message-passing system. The need to process messages dictates a basic application structure that is common to *all* PM applications that are written in C/2 or Macro Assembler/2, the OS/2-supported languages that support reentrant code. The structure of applications written in nonreentrant languages is different, and is described in "Programming without Application Window Procedures" on page 3-8. However, the intervening sections give you the background needed to understand the nonreentrant language support.

In OS/2 Presentation Manager, the structure of programs in reentrant and nonreentrant code is dictated by the requirement of an application to process messages from OS/2, other PM applications, and itself.

A reentrant PM application always contains two principal executable sections:

- A main procedure
- A window procedure.

each of which may call other procedures in the usual way.

The main procedure performs the following actions:

1. Initializes OS/2 system resources used by a PM application. The main procedure is the entry point to a PM application, like any C/2 program.
2. Creates an application input queue, to which OS/2 posts messages for the application.
3. Registers the class of the application-provided main window of the application, and its *window procedure*.
4. Creates the main window of the application.
5. Uses a message-processing loop to get messages from the queue, and to send them to the window procedure to be processed. The loop operates until a termination message is received.
6. Terminates OS/2 system resources used by a PM application.

The window procedure performs the following actions:

1. Processes every message in which the application is interested, and takes appropriate action.
2. Calls a system-provided default window procedure, to deal with any messages in which the application is not interested.

The basic structure of a PM application is as follows:

```
MainProc
  Initialize
  Create an input queue
  Register window procedure (WinProc)
  Create a window
  Loop until a termination message is received
     Get the next message
     Dispatch it to the window procedure
Terminate

WinProc
  Process messages for which procedure
  has to take specific action
  Pass everything else to system-provided
  default window procedure
Return
```

The Input Queue

In any supported programming language, a PM application that creates a window must call WinCreateMsgQueue (WMCMQ) to create an *input queue*. There is a WinDestroyMsgQueue (WMDMQ) call for when the application has finished with the queue.

The input queue acts as a buffer for user input from mouse and keyboard, so that no input is lost when the user is entering data faster than the application can process it. User input is strictly time-sequenced in the queue.

User input is processed one message at a time by the application calling the WinGetMsg (WMGET) function to pull a message off the application input queue, and calling WinDispatchMsg (WMDISP) to send the message to the window procedure for processing. When there is no message available, WinGetMsg (WMGET) goes into a wait state and does not return to the application until an event is available. WinPeekMsg (WMPEEK) can be used as another way of accessing the queue, particularly where the program needs to be aware of other mes-

sages in the queue, or where the application does not wait if there is no input available. WinPeekMsg (WMPEEK) does not wait if there is no input, and can also be used to leave the message on the queue.

Normally, input is processed in the order it appears in the queue. However, by *filtering* the messages, it is possible to process the input in an order determined by the application. Filtering involves using WinGetMsg or WinPeekMsg to check only for messages in a specific range. You must take care when filtering messages, because it is generally true that the processing of one input event changes the context for the next event. Taking messages out of order may produce a result not intended by the user. See the description of WinGetMsg in *Programming Reference: Volume 1* for details. In addition, another method of filtering is described in "Hooks" on page 13-19.

The application input queue allows the application to wait for all types of input simultaneously. The *application input queue is the only place in the application where user input is received.* This is demonstrated in the example programs, where there is a single call to WinGetMsg. The application can use the input queue to receive not only user input, but also other types of events.

Posting and Sending Messages

OS/2 and all PM applications pass messages by one of two operations, *post* and *send.* Post and send refer to distinct and separate operations. Do not confuse or interchange the terms.

OS/2 posts a message to an application as a result of an asynchronous event, such as user input, timers, and other OS/2 requirements. An application can call WinPostMsg (WMPOST) or WinPostQueueMsg (WCPQM) to post a message to itself or to another application. The difference is that you use WinPostMsg to post a message to a specified window, and it is placed on the input queue associated with the window. WinPostQueueMsg (WCPQM) posts the message to a specified queue. WinPostQueueMsg is used where you have a queue not associated with a window.

When a message is posted, it is placed on the input queue of the receiving task. When the message has been placed on the input queue, control returns to the posting task whether or not the message has been processed. The receiving task processes the message in its own time. Examples of posted input are messages that result from mouse or keyboard input. The message stays on the input queue until the application asks for it by calling WinGetMsg.

Sending a message is, in effect, a synchronous call to a window procedure. It is, therefore, like calling a function. Control does not return to the sending task until the window procedure has processed the message. The sending task does not have to take account of the location of the window procedure, or which process and thread own the window. OS/2 takes care of this. If a thread switch is required, it is performed by OS/2. You can send a message by calling WinDispatchMsg (WMDISP) or WinSendMsg (WMSEND). The difference is that the two calls present the message parameters in a different way. WinDispatchMsg (WMDISP) uses the whole message in a single parameter. WinSendMsg (WMSEND) uses all the message parameters as separate function parameters.

In the message processing loop of a reentrant program that has a window procedure, you use WinGetMsg (WMGET) to receive a message from the input queue: you use WinDispatchMsg (WMDISP) to send the message to your application window procedure. WinSendMsg (WMSEND) is more convenient when you construct a message and send it. In addition, many of the calls in the PM API implicitly send messages.

Posting and sending are both used in a PM application, as shown in Figure 3-6 on page 3-9.

Input Queue Function Calls

WinCreateMsgQueue (WMCMQ)
Creates a message queue

WinDestroyMsgQueue (WMDMQ)
Destroys a message queue

WinDispatchMsg (WMDISP)
Sends a message to a window

WinGetMsg (WMGET)
Gets a message

WinPeekMsg (WMPEEK)
Peeks a message, and can get it

WinPostMsg (WMPOST)
Posts a message to a message queue associated with a specified window

WinPostQueueMsg (WCPQM)
Posts a message to a message queue

WinSendMsg (WMSEND)
Sends a message to a window.

The Window Procedure

The application window procedure processes all input to the application's main window. It effectively defines the behavior of a window. The normal functions of the application take place in, or are called or started from the application window procedure. The window procedure can be invoked as a result of being sent messages by the main procedure, or can be invoked directly by OS/2. In other words, the window procedure is not merely sequentially executed, nor is it a conventional subroutine.

The system-provided window classes have their own system-provided window procedures in dynamic link libraries. (The classes are listed in "Window Classes and Object-Oriented Programming" on page 3-12.) When the user interacts with a window created by your application, whether it is a window for which your application provides a window procedure, or a system-provided window for which the system provides the window procedure, the system posts user input messages to your application's input queue. Figure 3-6 on page 3-9 illustrates OS/2 posting the messages to the top of the application queue, and an application using WinGetMsg (WMGET) to remove each message from the bottom of the queue. WinDispatchMsg (WMDISP) sends the message to the window procedure of the window identified by the window handle in the first parameter of the message. Each window procedure processes specific messages in a specific way, paints the window, determines the window's appearance and behavior, and changes its visual state as a result of user interaction. A window procedure can also be sent messages by other window procedures.

The body of the window procedure consists of a series of *case* statements. There is a case statement for each message type for which the window procedure is written to take a specific action. A *default* statement catches all the messages in which the application is not interested. The structure of the window procedure encourages you to structure your code. A message "drops through" the window procedure until it finds the case statement that matches its message type. The window procedure then takes the appropriate action. This can be a call to one or more OS/2 functions, or a call to a subroutine containing functions to perform processing for the particular message.

Under a case statement, the processing of a particular message can contain Winxxx functions that send messages to other windows or queues. A window procedure can send a message to itself, as shown in Figure 3-6 on page 3-9. Remember that this causes OS/2 to invoke the window procedure. The original instance of the window procedure cannot continue its processing until the instance of the window procedure that it has caused to be invoked returns. Therefore, the language in which the window procedure is written must support reentrancy, because the window procedure must be able to execute simultaneously with itself.

When the processing for a particular message is complete, the break statement breaks the processing out of the list of case statements and down to the return statement. The return statement returns control to the message processing loop. A window procedure has a return value, returned by the function WinDispatchMsg (WMDISP) that calls the window procedure. Again, the task switch occurs automatically, without the application code having to take an action.

If the message drops through the window procedure without finding a matching case statement, it finds the default statement at the end of the window procedure, and is processed by the default window procedure. The default window procedure provides the minimum default processing required for any message that is not processed by the application window procedure.

Programming without Application Window Procedures

FORTRAN and COBOL do not support reentrant code, so applications written entirely in these languages cannot contain application window procedures that process messages *sent* to windows of application-registered window classes.

However, you can write a mixed-language application, where your main program is in COBOL/2 or FORTRAN/2 but the window procedures for your application windows are in C/2 or MASM/2.

If you are programming entirely in COBOL/2 or FORTRAN/2, you can write a routine to process any messages that are *posted* to the window. To process messages that are sent to your main window, OS/2 provides a *language-support window procedure*. When an application registers a window

class with a NULL window procedure, the language-support window procedure intercepts messages that are sent to a window of that class. Consequently, the structure of FORTRAN and COBOL PM applications differs from that described in "Structure of PM Applications Written in a Reentrant Language" on page 3-6. (Although C/2 supports reentrancy, C/2 applications can also be written without application window procedures.)

Applications without application window procedures can consist of two principal executable sections:

- A main procedure
- A window routine.

each of which may call other procedures or routines.

The main procedure performs the following actions:

1. Initializes the OS/2 system resources used by a PM application. The main procedure is the entry point of the application.
2. Creates an application input queue, to which OS/2 posts messages for the application.
3. Registers the class of the application-provided main window of the application, with a null window procedure. The null window procedure tells OS/2 to provide a language-support window procedure, which intercepts messages that are sent to the window.
4. Creates the main window of the application.
5. Uses a message-processing loop that differs from the loop of an application that has a window procedure. Figure 3-7 on page 3-10 illustrates the message processing loop of a simple application. The call that gets the message does not immediately precede the call that dispatches the message. Instead, after the application gets the message, an IF statement or functional equivalent in the middle of the loop tests whether the target for the message is an application window. If it is, the application calls the application's window routine to process the messages. If the target is a window of a class for which a window procedure exists, the application dispatches the message to the window.

 Messages that are sent to the application window by other window procedures are intercepted by the language-support window procedure, described in the next subsection.
6. Terminates OS/2 system resources used by a PM application.

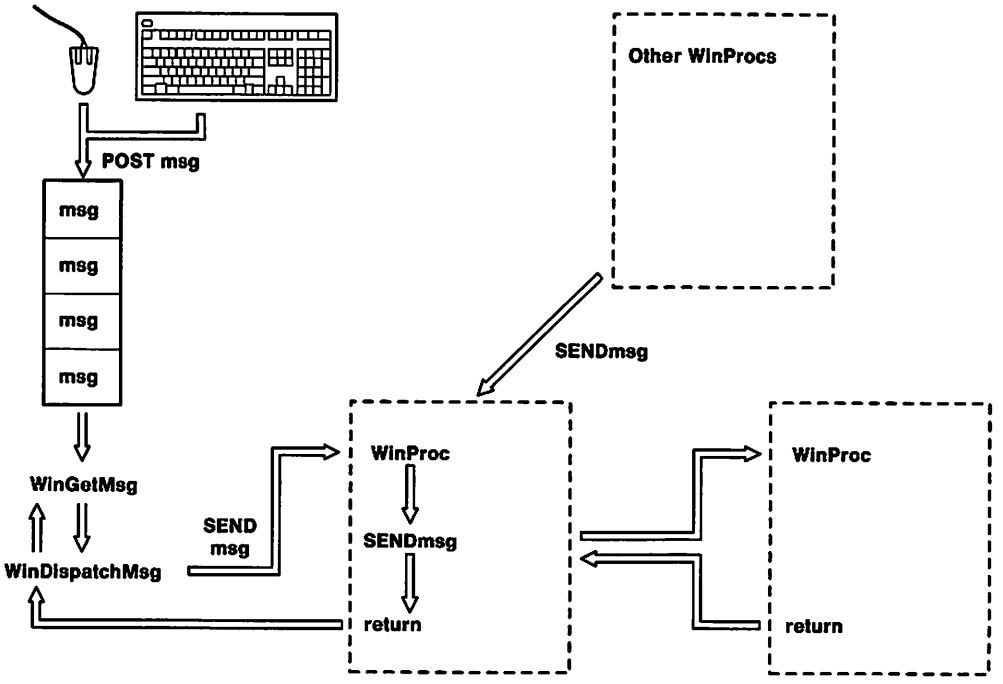


Figure 3-6. A Window Procedure Processes Messages

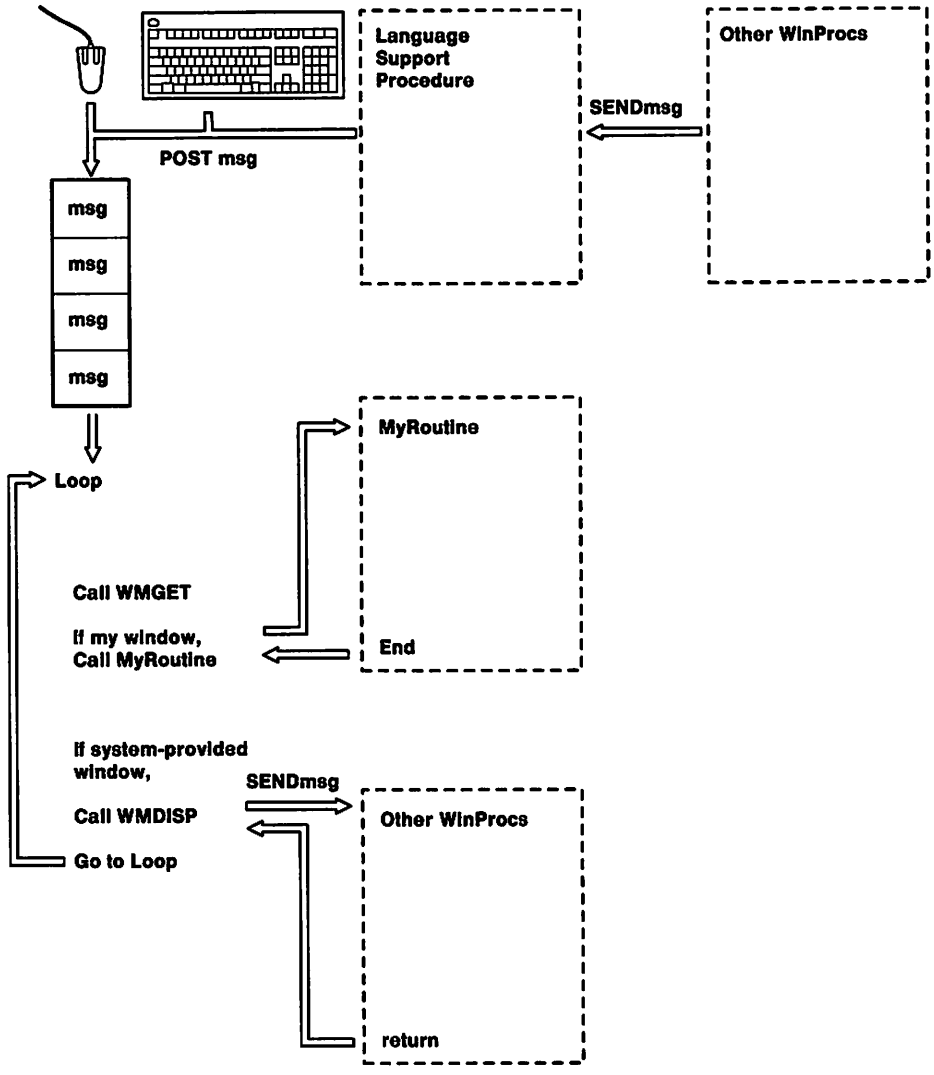


Figure 3-7. Message Processing for Nonreentrant Applications

The basic structure of a nonreentrant application is as follows:

```
MainProc
  Initialize
  Create an input queue
  Register null window class procedure
  Create a window
  Loop until a termination message is received
    Get the next message
    If destination window is my window,
      call MyRoutine
    Else dispatch to appropriate window procedure.
Terminate

MyRoutine
  Process posted messages for which routine
  has to take specific action.
  Pass everything else to system-provided
  default window procedure.
End
```

Note that the window routine MyRoutine is similar to a window procedure in that it tests for particular message types, and processes them appropriately. Also similarly to a window procedure, MyRoutine calls the default window procedure to deal with any messages in which the routine is not interested. However, unlike a window procedure, MyRoutine processes *posted* messages only. It is never sent messages, and is never directly invoked by OS/2, or by itself.

The Language Support Window Procedure

The language-support window procedure processes messages sent to a window of a window class that has no window procedure. For most messages, the language-support window procedure takes the same action as the default window procedure. However, for the important sent messages the language-support window procedure responds by posting them to the application's input queue. To prevent any closed loops occurring, the procedure gives the messages different names when it posts them. For example, when the language-support window procedure is sent a WM_PAINT message it posts a WM_PPAINT message.

The messages, and their posted counterparts, are as follows:

WM_ACTIVATE
: Sent when an application causes the activation or deactivation of a window. A **WM_PACTIVATE** is posted.

WM_PAINT
: Sent when a window needs repainting. A **WM_PPAINT** is posted.

WM_CONTROL
: Sent to the owner of the control when an event changes the control. A **WM_PCONTROL** is posted.

WM_SETFOCUS
: Sent when a window is to receive or lose the input focus. A **WM_PSETFOCUS** is posted.

WM_SIZE
: Sent when a window changes its size. A **WM_PSIZE** is posted.

WM_SYSCOLORCHANGE
: Sent when a change is made to the system colors by the WinSetSysColors (WISCOL) function. A **WM_PSYSCOLORCHANGE** is posted.

The delay in receiving sent messages that are posted on and retrieved from the queue is not large enough to be significant.

PM Limitations in COBOL and FORTRAN

A window without a window procedure does not have the same control over the user interface as does a window with a window procedure. Any PM facilities that depend on sent messages (other than the posted-on subset in the previous section) are not available to a window without a window procedure. This is because the window never receives the messages, and so cannot act on them.

Facilities that are **not** available are:

Dynamic Data Exchange
: A protocol for applications to send messages to each other, that establish and implement the exchange of data between the applications, without directly involving the user.

Window subclassing
: A technique for intercepting messages intended for another window procedure.

Owner-draw items
Drawing your own graphics for choices in menus or listboxes depend on a sent message.

In addition to the above, some PM facilities have restrictions in a nonreentrant language:

Clipboard
This is an area of storage that can be used as an intermediate holding area in the editing operations cut, copy, or paste. Nonreentrant applications can use this area, but they cannot own it. A facility known as delayed rendering is also not available. See Chapter 28, "User-Driven Data Exchange – The Clipboard" for details.

Dialog boxes
This is a class of window that is processed differently in nonreentrant languages. See Chapter 7, "Dialog Boxes and Controls" for details.

Notification messages
These are sent to the application as a result of calls that the application itself makes. For example, the calls that create, destroy, move, size, hide, or show a window send notification messages to the window. With the exception of WM_SIZE, which the language-support window procedure posts on, a window without a window procedure does not receive such messages. However, such a window can, immediately after the call that causes the notification message, call the application code that would correspond to the processing of the notification message.

An alternative technique, in code that depends on a particular window state, is to compare a copy of the saved state of the window with the window. If any changes are detected, the application can call the code to process the state change.

Storage allocation
Some PM functions require preallocation of storage by the application before invocation. PM uses this storage to return the result of the function (for example, a text string). In COBOL and FORTRAN, programmers must allocate this storage as static storage.

Other facilities that are not available are:

The Heap Manager
A facility for the management of memory segments and memory objects, is also not available to COBOL and FORTRAN programs.

Hooks
These are intended for use by system programmers working at assembler level (or just above) only.

Areas in which additional coding work is required (compared with C/2, for example) are:

Structures
Structure-manipulation capabilities are not native to the COBOL and FORTRAN languages, so they are provided by the PM COBOL/FORTRAN bindings code.

Large PM constants
A few PM constants have values that are larger than those directly supported by the COBOL and FORTRAN languages. PM utility functions are provided so that your program can manage the use of such constants.

Mixed Language Programs

Many of the limitations listed in the last section can be overcome by writing a mixed-language application, where your main program is in COBOL/2 or FORTRAN/2 but the window procedures for your application windows are in C/2 or MASM/2. In this case, the code for your application windows resides in a dynamic link library. How your application links to that code is determined by the window class of the windows, described in the next section.

Window Classes and Object-Oriented Programming

Each window that belongs to a specific window class uses the window procedure associated with that class. The window procedure processes message input and defines the appearance and behavior of a window class. An application can create one or more individual windows that belong to the same window class. Because each individual window of the same class is processed by the same window procedure, they all behave in the same way. In addition, coding overhead is reduced because many windows can result from *one* window procedure.

There are two types of window class: public and private.

Public Window Class

A public window class is one that has a reentrant window procedure that is registered and resides in a dynamic link library. It can be used by *any* process in the system to create windows.

OS/2 provides several preregistered public window classes. You can specify the system-provided window classes by using OS/2-provided symbolic identifiers that have the prefix WC_ :

WC_BUTTON
Button.

WC_ENTRYFIELD
Entry field.

WC_FRAME
Frame window. This is a composite window class that can contain windows of many of the other window classes as children.

WC_LISTBOX
List box.

WC_MENU
Menu.

WC_SCROLLBAR
Scroll bar.

WC_STATIC
Static control, such as output-only text.

WC_TITLEBAR
Title bar.

With the exception of WC_FRAME, the system-provided window classes are known as *control window classes*, because they give the user a simple means to control specific types of interaction. For example, the WC_BUTTON class allows single or multiple selection of choices. These "ready-made" windows conform to the IBM Systems Application Architecture Common User Access definition. They are specifically designed to provide function that meets most of your needs for a graphics-based standard user interface. The PM sample applications described in this guide make extensive use of the WC_ class windows.

You can use public window classes in your application without registering them, because OS/2 has already registered them.

Because the window procedures of the system-provided windows are reentrant, they can be used by reentrant or nonreentrant programs. However, nonreentrant programs can only process messages that are posted by the system-provided window classes.

The Presentation Manager constants files for COBOL and FORTRAN contain the system-provided window classes in the form of string constants. For example, a button control is specified with `WC-BUTTON = '#3'`.

An application can create its own public class in a dynamic link library, but it is done in a special way at system initialization time. See "Registering a Public Window Class" on page 13-9 for details.

Private Window Class

A private window class is one that your application registers for its own use, and is available only to the process that registers it. A private class has an application-provided window procedure either in the application's .EXE file or in a DLL file. See "Registering a Private Window Class" on page 13-9 for details. A private window class is deleted when the process that registers it terminates.

When you register a window class, you actually register the window procedure used by the window class. (Applications without window procedures register a NULL window procedure.) This is done using the WinRegisterClass (WMRCL) call. You can subsequently create a window belonging to the registered class.

Object-Oriented Programming

Each window of a registered class can have private data, called *window words*. (See "Using Window Words" on page 13-14 for details.)

In addition, you can define window *resources* such as text strings and menus, icons, bit maps, pointers, fonts, and even child windows in files separate from your source code. You define or refer to the resources in a *resource file* for the application.

Using windows to provide separate functions and data in subdivisions of the screen encourages you to structure your program. Each OS/2 window, its data, its resources and its window procedure are a programming object. For each relevant message, the programming object responds in a defined way to the event represented by the message. Several responses are possible. For example, an application can change the data displayed by the window,

internal data, or external data such as a disk file. The PM programming style is object-oriented and event-based.

You can create one or more *object windows* that are never actually displayed, but exist as invisible programming objects that process data. See "Creating an Object Window" on page 13-14 for details.

Class Styles

A public or private window class can have a set of OS/2-defined properties called *class styles*. Class styles are 32-bit values defined in OS/2 and have the prefix CS_ . They define, among other things, the actions that OS/2 performs when repainting, moving, or sizing operations affect a window belonging to the registered class. Class styles give you the ability to "finetune" the behavior of a registered class, and are supplied as parameters to WinRegisterClass (WMRCL). The class styles follow:

CS_CLIPCHILDREN
Prevents the window from painting over any of its child windows.

CS_CLIPSIBLINGS
Prevents the window from painting over any of its sibling windows.

CS_FRAME
The window behaves as a WC_FRAME window, regardless of its class.

CS_HITTEST
A WM_HITTEST message is sent to the window, before a mouse message is sent, to test whether the window wants the mouse message.

CS_MOVENOTIFY
A WM_MOVE message is sent to a child window that has this class style, when its parent is moved.

CS_PARENTCLIP
Clips the window to the boundaries of its parent. Generally, a window of this style should not draw outside its window rectangle.

CS_PUBLIC
Registers a public window class.

CS_SAVEBITS
Saves the underlying screen image, when a window of this class style is made visible.

CS_SYNCPAINT
The window is synchronously repainted. (This is described in "Output in Windows" on page 3-18.)

CS_SIZEREDRAW
Redraws the window when it is sized.

Window Subclassing

In a technique called *window subclassing*, a reentrant application can intercept messages before they are received by the window procedure of a public or private window class. The application can pass the messages to a window procedure of its own, where it can discard them, or change them, or process them itself, or pass them on to the original window procedure. In reusing the original window procedure, you are saved the effort of coding an entire window procedure. See "Subclassing a Window" on page 13-9 for details.

Creating Windows

You can create a window explicitly using the WinCreateStdWindow (WICRTS) or WinCreateWindow (WICRT) calls. WinCreateStdWindow creates a single individual instance of a WC_FRAME class window, which can create and contain an application-registered window. You can use it to create the main window of an application, at the top of the application's logical hierarchy. A WC_FRAME window is also known as a *standard window*, and is described in Chapter 5, "The Standard Window." WinCreateWindow creates a single individual instance of any WC_ class or application-registered window class.

Also, you can create a window implicitly by loading a definition from a window template in memory or a resource file, as described in Chapter 7, "Dialog Boxes and Controls."

The total number of windows that the system can support is approximately 1200. The recommended maximum number of windows that each application should create is 100. In the unlikely event that the system runs out of windows, a NULL window handle is returned by the call creating the window.

Window Styles

A window of a public or private window class can have a *window style*. The window style is a 32-bit value that controls the appearance and behavior of an *individual* window. You specify the window style by combining various style bits with an OR operator. There are several standard style bits that you can specify for an individual window of *any* class. The standard window style bits are restricted to the high-order 16 bits of the window style. The standard window-style bits are defined in OS/2 as symbolic identifiers with the prefix WS_. They follow:

WS_CLIPCHILDREN
Prevents the window from painting over any of its child windows.

WS_CLIPSIBLINGS
Prevents the window from painting over any of its sibling windows.

WS_DISABLED
Disables the window so it cannot receive user input. The default is enabled.

WS_MAXIMIZED
Creates the window maximized.

WS_MINIMIZED
Creates the window minimized.

WS_PARENTCLIP
Clips the window to the boundaries of its parent.

WS_SAVEBITS
Saves the underlying screen image, when a window of this style is made visible.

WS_SYNCPAINT
For windows of class style CS_SYNCPAINT, you can turn this window style on and off. When the style is on, the window is synchronously repainted. (This is described in "Output in Windows" on page 3-18.)

WS_VISIBLE
Creates the window visible. The default is invisible.

Do not confuse the window style with the class style (CS_) which controls the appearance and behavior of an entire window class. You may have noticed that some of the window styles have corresponding class styles. If the CS_CLIPCHILDREN, CS_CLIPSIBLINGS, CS_PARENTCLIP, or CS_SAVEBITS class style is set, you get a window of the corresponding window style regardless of the WS_ setting.

The above list omits the two WS_ settings WS_GROUP and WS_TABSTOP, that apply only to the particular type of window described in Chapter 7, "Dialog Boxes and Controls."

In addition to the above window styles that apply to windows of *any* class, there are window styles that apply specifically to windows of each class. For example, some window styles apply only to windows of the WC_FRAME class, some apply only to windows of the WC_BUTTON class, and so on. These are described under each window in this guide.

The Input Process

This section explains the process by which input reaches an application through the input queue. It explains the internal design of the OS/2 input mechanism for PM applications, and clarifies the process and functions that are available to your application.

The system converts user input from the mouse and the keyboard into a series of messages, which it places in a single *system queue*, in a strictly time-ordered serial sequence. This includes the user input for *all* applications. This is shown in Figure 3-8 on page 3-16.

Messages are removed from the system queue by the *input router*. The input router decides which application's input queue receives the message.

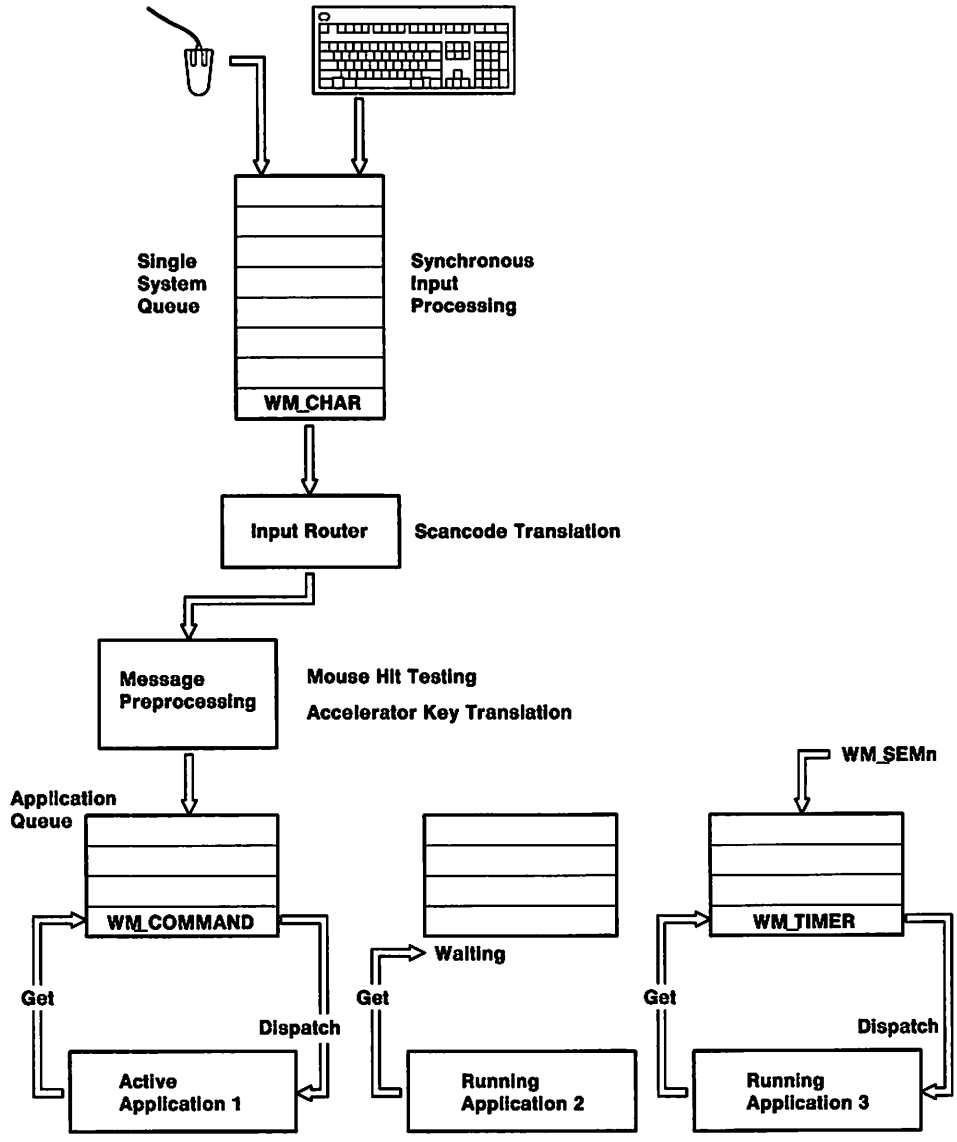


Figure 3-8. Windowed Application Input Process

Keyboard Input

Keyboard messages initially have only a raw scancode indicating which key was pressed or released. The input router translates the scancode into a character code or virtual key value and maintains the shift status, which is also placed in the translated keyboard message. The raw scancode is also present in the translated keyboard message, but this value is hardware-dependent. The translated character code or virtual key value is hardware-independent.

OS/2 posts a WM_CHAR message to the application input queue associated with the input focus window, unless the key is defined as an accelerator key. In this case, OS/2 translates the WM_CHAR message into the message type specified in the resource file.

Mouse Input

OS/2 usually directs mouse input, in the form of messages such as WM_BUTTON1DOWN, to the queue associated with the window below the pointer when the message is generated. However, if the window below the pointer belongs to a window class that has the CS_HITTEST class style, OS/2 *hit tests* it by sending it a WM_HITTEST message to check whether it wants to receive mouse messages. The window can respond in one of the following ways:

1. Indicate that it will accept the mouse message. In this case, the message is passed on normally.
2. Indicate that the message should be discarded. In this case, it is discarded and not sent to any window.
3. Indicate that it does not want to handle the message. In this case, the next window below in the z order is sent a WM_HITTEST message. If it indicates that it will accept the message, the next window receives the mouse message. This is often used when you do not want a child window to handle mouse input directly, but want its parent to process it instead.

Capturing Mouse Input

Mouse input can go to a window other than the one below the pointer in one other circumstance. A window can *capture* the mouse. When a window captures the mouse, the window unconditionally receives *all* mouse input. Typically, a window should capture the mouse when it needs to track an operation that involves moving some visual data outside its own area. An example of this is dragging a display object across the screen, as a means of copying or moving the data to another window.

The WinSetCapture (WMSCAP) call captures *all* mouse input, regardless of the pointer position. The call is also used to release the mouse. When WinSetCapture is called, a WM_MOUSEMOVE message is posted to the application, regardless of whether the mouse actually moved. This lets your application respond if it wants to, for example, by changing the shape of the pointer.

You can use the WinTrackRect (WITREC) call to draw a tracking rectangle. There is also a WinShowTrackRect (WISTR) call that makes the tracking rectangle invisible or visible.

The Input Queue

The application input queue is created by the WinCreateMsgQueue (WMCMQ) function. This must be called by a thread that receives input. The size of the queue is determined by an input parameter to WinCreateMsgQueue (WMCMQ). The application queue is not only used for user input, but is also used for other kinds of messages:

- WM_PAINT messages, which indicate that part or all of a window needs repainting.
- WM_TIMER messages, which are generated when a timer started by the WinStartTimer (WISTAT) function expires.
- WM_SEMn messages, which can be posted to an application to indicate that some kind of application-related event has occurred. The n is in the range 1 through 4, and control the relative priority of the message.
- Inter-application messages. Messages of a general kind can be posted to an application. Messages of a specific kind, for example, containing the handle of some shared data, can be posted to an application. These messages appear on the application input queue.

To get input, applications must call the WinGetMsg (WMGET) function (or the similar WinPeekMsg (WMPEEK) function). Typically, all threads that handle user input make a call to WinGetMsg, as shown in Figure 3-8 on page 3-16. It is normal for only *one* of the input-handling threads to be executing. The rest are waiting in their WinGetMsg call for the next input message directed their way.

Message Priority

WinGetMsg (WMGET) obtains the next available input message from the application input queue. If more than one message is available, the messages have the following priority:

1. WM_SEM1 message
2. Inter-Application message
3. User input message
4. WM_SEM2 message
5. WM_TIMER message
6. WM_SEM3 message
7. WM_PAINT message
8. WM_SEM4.

If none of these messages is available, the thread is suspended within WinGetMsg until a message becomes available.

Synchronous Processing

Although only one input thread can be processing *user* input, because of synchronous processing, the input threads of other applications can be processing other kinds of input events. For example, moving the active window across the screen can

generate a series of WM_PAINT messages for the other windows on the screen. The threads to which those windows belong run and receive the WM_PAINT messages from WinGetMsg, although they cannot receive user input. Therefore, the background windows are updated to their proper appearance, even though the user is not interacting with them.

A PM application processes user input (but *not* other types of input) *synchronously*. Therefore, a message is not processed until the previous message has been completely processed. Completion of processing is indicated when the thread that received the message calls WinGetMsg again.

The serialization of user input messages on the system queue ensures synchronized processing. In a rapidly changing screen, the processing of one message can affect the meaning of the next message. For example, a keyboard or mouse message can change the active or input focus window. Serialization ensures that subsequent input is directed to the window for which it is intended. With type-ahead and mouse-ahead, user input to a series of windows can line up on the system queue. A five- or six-character sequence can take the user through an action bar, a pull-down, and then fields in a pop-up window displayed as a result of selecting a choice in the pull-down. An expert user can type the whole sequence before either the action bar *or* the pull-down is activated.

The fact that more input is in the queue for the application is actually used for optimization by the system. In this example, if the user types fast enough, neither the pull-down nor the pop-up window are shown on the screen. The function implied by the key sequence is executed without any relatively slow screen updates taking place.

User Responsiveness

A window procedure always runs in the thread that creates the window. This is true for the window procedures of both private and public window classes. With synchronous processing of user input by an application, no further user input messages can be pulled from the application input queue until the current user input has been completely processed. This guarantees correct processing in an environment that has the potential for rapid change. Synchronous processing means that the user is kept waiting if an application takes a long time to process a user input message. The application takes no notice of any further user input until the processing is finished.

Another feature of the system queue is that no user input messages are routed to any application input queue until all previously routed user input messages have been processed.

The input thread must stay responsive to the user. An application that does not stay responsive is "badly behaved." As a result, the user can be locked out for some time, and is unable to interact with the application *or* the rest of the system.

A "well-behaved" PM application completes processing of user input within 0.1 to 0.5 seconds. To do this, the input-handling thread must call WinGetMsg (WMGET) or WinPeekMsg (WMPEEK) within this range of time, after returning from the previous call to one of these functions.

If your application has some processor-intensive work to do, it should use the asynchronous processing capabilities of OS/2 to become a multithreaded application. One message-queue thread is used for user input and other threads for the lengthy processing. The design of such applications is discussed in "Multithreading Considerations for PM Applications" on page 34-4.

Output in Windows

Each window has data placed into it; it is a place to display output data. An application is responsible for painting the data for the window classes that it registers and creates. This data can be graphics text or pictures, as described in detail in "Part 4. Presentation Manager Applications – Using the Graphics Programming Interface." It can be fixed-size alphanumeric text, as described in detail in "Part 6. Alphanumeric Text Applications – User Input/Output." It is not normally necessary for the application to paint the system-provided window classes. The OS/2 window procedures for those window classes do the painting.

The Presentation Space

Before a PM application can display alphanumeric text, graphics text, or pictures in a window, the application must create a *presentation space*. A presentation space is an area of storage in which an application assembles a picture before it displays it

in a window or prints it. There are two general types of presentation space:

- Alphanumeric Vio (AVIO) presentation space
- Graphics presentation space.

A Vio presentation space displays fixed-size alphanumeric text in a window, in the style of the IBM 3270 display text. It is described in "Part 6. Alphanumeric Text Applications – User Input/Output."

A graphics presentation space displays graphics. A presentation space (PS) can be smaller or larger than the window in which it is displayed. If larger, it is clipped to the window boundary.

A choice of coordinate systems is available, by options on the GpiCreatePS (GIPCRT) call.

A graphics presentation space can be created explicitly or implicitly. There are three types of graphics presentation space: normal PS, micro PS, and cached micro PS. A normal PS or a micro PS can be initially created explicitly by using call GpiCreatePS (GIPCRT). Which PS you use depends on the complexity of the graphics you are drawing. A micro PS is used for simple drawing operations, where you do not need to store long-term graphics data. A normal PS is required when using certain Gpi functions. See "Part 4. Presentation Manager Applications – Using the Graphics Programming Interface" for details.

In the processing of the WM_PAINT message, an application begins painting using WinBeginPaint (WIBPNT). If a presentation space has already been created, specify the handle of the existing presentation space in the second parameter. This action obtains the presentation space in which to draw. If no presentation space has been created, specify NULL in the second parameter. WinBeginPaint obtains a micro PS from a cache of micro PSs kept by OS/2. This is the implicit form of presentation space creation. (A cached micro PS can also be initially obtained using WinGetPS (WPGPS) or WinGetScreenPS (WPGSPS)). A micro PS is contained inside the window boundary, and uses window coordinates. WinBeginPaint (WIBPNT) returns a presentation space handle. This contains a number used by OS/2 to identify the presentation space uniquely. The handle is subsequently used as a parameter on most graphics calls.

Visible Region

The portion of a presentation space visible in a window is called the *visible region* of the window. When you paint graphics in a window, OS/2 automatically clips the graphics data to the window's visible region.

The visible region of a window is the presentation space clipped to the boundary of the window and the boundaries of any overlying windows. Overlying windows can be child windows and windows belonging to other applications. Overlying windows restrict the visible region to less than the full window. Data outside the visible region of a window is clipped away and cannot be seen. This happens at the window boundary. You can think of the visible region as a set of one or more rectangles, which are not necessarily contiguous.

For example, Figure 3-9 shows window 2, containing a picture of an airplane. The central portion of window 2 is overlaid by another window, window 1. The non-contiguous rectangles visible on each side of window 1 comprise the visible region of window 2.

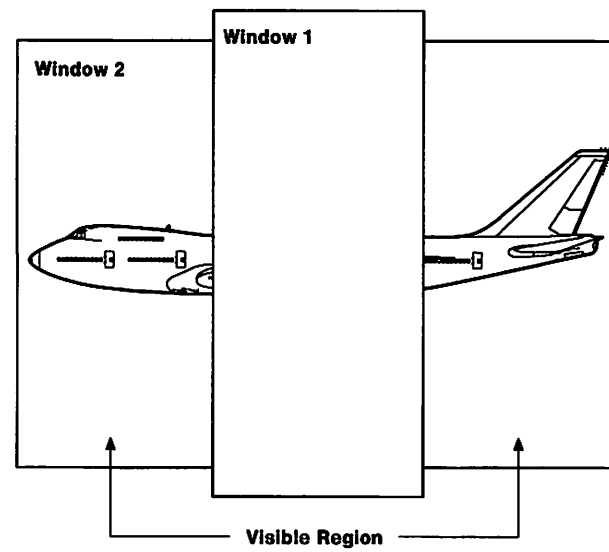


Figure 3-9. Visible Region of a Window

Update Region

Associated with a window that requires repainting is an area of dynamic storage called an *update region*. When an overlying window is removed or moved, the underlying part of a window is uncovered. It is visually "invalid," or incorrect, and must be repainted. The system can add the invalid rec-

tangle to the update region – one or more rectangular areas (not necessarily contiguous) that are associated with a window and are visually "invalid" or incorrect. For example, Figure 3-10 shows the part of window 2 that the system adds to its update region when the user removes window 1. The application must repaint the middle part of the airplane.

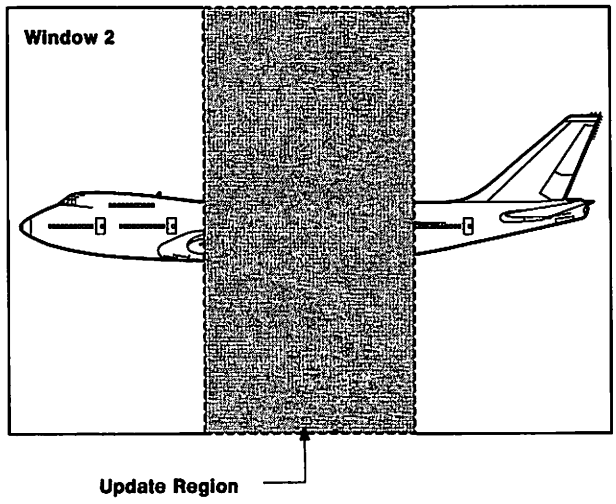


Figure 3-10. Update Region of a Window

Adding one or more rectangles to the update region is called *window invalidation*. The WinQueryUpdateRegion (WIQURG) call obtains the handle of the update region. The WinQueryUpdateRect (WIQURC) call obtains the address of a structure containing the coordinates of the smallest rectangle that encloses (bounds) the update region.

In addition to the window invalidation performed by the system, an application can explicitly add rectangles or regions to the update region using the WinInvalidateRect (WINVR) and WinInvalidateRegion (WINVRG) calls. For asynchronous windows, rectangles and regions can be removed from the update region using the corresponding calls WinValidateRect (WIVREC) and WinValidateRegion (WIVRGN).

Synchronous and Asynchronous Painting

An application can choose to repaint a window synchronously or asynchronously. The synchronous – asynchronous choice is controlled by the CS_SYNCPAINT and WS_SYNCPAINT styles.

Synchronous Painting

In synchronous repainting, the system sends a WM_PAINT message to the application at the moment that an operation invalidates the window. The operation does not complete until the application has processed the WM_PAINT message. An application usually repaints a synchronous window when it receives the WM_PAINT message.

A window that is synchronously updated is called a *synchronous-paint* window. Use a synchronous-paint window when you want a window to be redrawn quickly.

Asynchronous Painting

With asynchronous painting, the system does *not* tell the application that the window is invalid when other messages are on the application's input queue. Instead, it accumulates the rectangles invalidated by this *and any subsequent operation* in the window's update region. When there are no messages in the queue, and any window associated with the current queue has a non-empty update region, a call to WinGetMsg (WMGET) or WinPeekMsg (WMPEEK) returns a WM_PAINT message for that window. The system does not place the WM_PAINT message in the queue. Because the system can accumulate multiple invalidations in the update region, a single WM_PAINT message can be returned as a result of more than one invalidation.

Where more than one asynchronous window is associated with an input queue, and more than one window is invalid, WM_PAINT messages are returned in the order from the top of the logical window hierarchy to the bottom.

A window that is asynchronously updated is called an *asynchronous-paint window*. You should use an asynchronous-paint window when a window can take a long time to be painted. By delaying redrawing until no messages exist, an application can keep up with a user who quickly types or uses the mouse.

When an *asynchronous*-paint window is the parent of a synchronous-paint child window and they are invalidated by a single operation, the painting of the children is deferred until the asynchronous-paint parent processes the WM_PAINT message.

Repainting Synchronous or Asynchronous Windows

In your application's processing of the WM_PAINT message, you supply the presentation space handle to the WinBeginPaint (WIBPNT) call. The system clips the presentation space to the clipping region of the window. The clipping region is the region formed by the intersection of the visible region and the update region of the window. Therefore, if your application repaints the entire presentation space in response to window invalidation, only the invalid parts are repainted on the screen. WinBeginPaint also obtains the address of a data structure containing the coordinates of the smallest rectangle that encloses the update region. An application can use this information to optimize the extent to which it redraws its presentation space. WinBeginPaint (WIBPNT) resets the update region to NULL. The system assumes that any drawing following WinBeginPaint restores the contents of the window to a fully valid state.

When drawing is complete, the application calls WinEndPaint (WIEPNT). In the processing of this call, the system updates any synchronous-paint windows that are children of the window being repainted.

Incremental Repainting Asynchronous-Paint Windows

An application can incrementally repaint an asynchronous-paint window one area at a time. While a window is invalid (has a non-null update region), WM_PAINT messages are returned by WinGetMsg (WMGET) or WinPeekMsg (WMPEEK). The application can use WinQueryUpdateRect (WIQURC) or WinQueryUpdateRegion (WIQURG) to obtain a rectangle or region that requires repainting. The calls do not affect the update region. Based on this information, one or more parts of the window can be repainted, and the corresponding part or parts of the update region validated by WinValidateRect (WIVREC) or WinValidateRegion (WIVRGN).

Repainting Obscured Windows

A call WinIsWindowShowing (WIQSHO) has been introduced which enables the application to determine whether a window is visible before updating the logical video buffer. If the return from the call says the window is not visible the application has the option not to repaint at that time. If this method of avoiding unnecessary repainting is used the call must be issued each time new information is received. If this is not done invalid screen content could result.

An example of when this can be useful is when information is being received from a host using a terminal emulator. The time taken to receive the information into the logical video buffer and then update the screen can be significant and reducing the number of times this has to be done can increase drawing performance.

Device Contexts

When your program outputs data to a device, a screen for example, it must use a ***device context***. A device context is a device-dependent definition of the device characteristics and the software that runs the device (the device driver). A device context can be a definition of one of the following:

Screen/window
: The display is used implicitly by all windows. All window device contexts use the same physical device, but each has its own logical device context to which any output is directed.

Queued output
: In a multiprocessing system like OS/2, output queues are the normal method for applications to direct data to a hard-copy device. The data does not pass directly to the device, but is first placed in a spool queue on a disk. The spool files are processed by a spool processor, which sends one file at a time to the device.

Direct (non-queued) output
: Direct output device contexts allow an application to drive a hard-copy device directly, without the data passing through a spool file. Normally, an application does not do this, but uses a queued output device context. However, a spool processor is an example of an application that drives a hard-copy device directly, and uses a direct-output device context. Specialized applications may have dedicated hardware that should not be shared with other applications under OS/2. In this case, a direct output device context is appropriate.

Memory bit map
: A memory bit map device context is used for creating and manipulating bit maps. A memory device context always has a format

that is compatible with a specific hardware format for a bit map.

Metafile
A metafile device context is used for saving graphics in a file known as a metafile, that has a standard published format.

Information
It is possible to open a device context for the purpose of obtaining information only. This saves storage and time if information only is required.

Before your application can send graphic or alphanumeric data to an output device, metafile, or bit map, it must create a device context, create a graphics or alphanumerics presentation space, and associate it with the device context. Drawing to the presentation space causes output to the device defined by the associated device context.

If the application subsequently associates the presentation space with a different device context, the same data can be directed to a different device.

Window Painting Calls

WinBeginPaint
Obtains a presentation space whose associated update region is set ready for drawing in a specified window

WinEndPaint
Indicates that the drawing of a window is complete, generally in the processing of WM_PAINT

WinExcludeUpdateRegion
Subtracts the update region of a window from the clipping region of a presentation space

WinGetPS
Obtains a cache presentation space

WinGetScreenPS
Obtains a presentation space that can be used for drawing anywhere on the screen

WinInvalidateRect
Adds a rectangle to a window's update region

WinInvalidateRegion
Adds a region to a window's update region

WinLockVisRegions
Locks or unlocks the visible regions of all the windows on the screen, preventing them from changing

WinOpenWindowDC
Opens a device context for a window

WinQueryUpdateRect
Obtains the rectangle that bounds the update region of a window

WinQueryUpdateRegion
Obtains the update region of a window

WinReleasePS
Releases a cache presentation space obtained using WinGetPS or WinGetScreenPS

WinUpdateWindow
Forces the update of a window and its associated child windows

WinValidateRect
Subtracts a rectangle from the update region of an asynchronous-paint window

WinValidateRegion
Subtracts a region from the update region of an asynchronous-paint window.

Conforming to Common User Access

Common User Access is one part of Systems Application Architecture, a set of selected software interfaces, conventions, and protocols. These serve as a common framework for the design, development, and portability of applications across the three IBM major computing environments:

- Personal Computer (PS/2 or IBM Personal Computer AT under OS/2)
- System 370 (TSO/E under MVS/XA, and CMS under VM)
- AS/400.

Common User Access is a set of rules that define the way information is presented on a screen, and the techniques for the user to interact with the information. The rules are documented in the publication *Systems Application Architecture Common User Access Advanced Interface Design Guide*. If you want to ensure that the user interface of your application conforms to Common User Access, you should read that publication, and apply its principles to your application.

In the chapters in this part of the guide, you will see how you can provide the user interface of your application, using the system-provided window

classes. These are standard visual controls that let the user make one or more choices, or enter text. They are specifically designed for user dialog, and conform with Common User Access.

The aim of Common User Access, through use of a consistent user interface, is to provide ease of learning, ease of use, and the efficient transfer of acquired familiarity across a whole range of programs. The user shell itself has a user interface that is based on Common User Access. The OS/2 API provides you with facilities that make it easy to produce applications whose user interfaces also conform with Common User Access.

A consistent interface benefits users and application designers, saving them time and money. Users take less time learning to use an application, and may take less time using it. An additional benefit to users is reflected in their attitudes. A consistent interface reduces users' frustration levels, increases their feelings of accomplishment, and makes them feel more comfortable with the system. A consistent user interface benefits application designers by defining common building blocks for an interface through standardized interface elements and interaction techniques. These building blocks let you create and change applications more easily and quickly. Because the same panel can be used across many systems, you can reuse panel designs.

Common User Access applies various user terms to the visual elements of the user interface. These occasionally differ from the terms used by OS/2 to describe the programming objects used to implement those visual elements. As far as possible, this book uses Common User Access terms to refer to the visual elements, and OS/2 terms to refer to the programming objects. This chapter, therefore, gives you a bridge between the two sets of terminology.

Common User Access describes user interface panels for two separate application types, classified according to the means used to display output on the screen. The types are character applications and graphics applications.

In OS/2, character applications display dialogs using fixed-size alphanumeric text only. This is displayed by your application using text that is either explicitly coded in your application, or read by your application from files created by you. Alphanumerics for OS/2 applications are described in "Part 6. Alphanumeric Text Applications – User Input/Output."

Graphics applications can use the graphics programming interface to paint graphics text and pictures in windows, as described in "Part 4. Presentation Manager Applications – Using the Graphics Programming Interface." For dialog with the user, graphics applications can use the system-provided windows described in this part of the guide.

Primary, Secondary, and Pop-up Windows

Common User Access defines four types of windows: primary, secondary, dialog box, and message box. These terms describe windows in terms of the function that the windows provide to the user.

Every program should have a primary window. Primary windows display the *primary* function of an application in a panel or series of panels. For example, in a text editor, the primary window contains the text being edited. In a spreadsheet, the primary window contains a spreadsheet.

Secondary windows (if required – they are not mandatory) display function that is *secondary* to the primary function. For example, secondary windows provide:

- A parallel view of some or all of the data in the primary window.
- Information that assists the function in the primary window, for example, help information.
- Options that change what the user sees in the primary window. For example, in a business graphics application, a secondary window might contain a panel that lets users change the chart type, while the primary window displays the chart.

The user can switch back and forth between primary and secondary windows.

In OS/2, primary windows are usually implemented using a system-provided standard window. A standard window can have a title, frame, a menu of choices, and a central area called the client area. The window title associates the window with its application. You can extend the dialog by associating a pull-down with one or more of the action bar choices. The client area is enclosed by the frame and is reserved as a place in which the appli-

cation can create its own windows and place output. The standard window usually can be sized and moved by the user, and can also contain the means for the user to scroll the data in the client area. The standard window is described in Chapter 5, "The Standard Window."

Secondary windows containing help or options can be implemented using a standard window, with at least a title and a border, and usually can be sized and moved. If you want the user to be able to move a secondary window outside the boundary of a primary window, you should create both windows as children of the desktop. Alternatively, if you want the secondary window always to be contained within the primary window, create the secondary window as a child of the primary window.

The function in primary and secondary windows can be displayed using graphics text, pictures, or both, painted by the application, or by using a system-provided dialog box window class.

You can further extend the dialog by displaying one or more dialog boxes when a choice is made from a pull-down. When a pull-down choice is followed by three periods in a row (... known as an ellipsis), it indicates that one or more dialog boxes are associated with the choice, and will appear when the user selects the choice.

Message boxes are used to display messages to the user.

A dialog box can contain the following, as child windows:

- A variety of the system-provided WC_ control windows with which the user can indicate a particular choice of function
- Static text to accompany the controls
- Icons and bit maps.

Although you specify the dialog controls in your application or in a resource file, they belong to the system-provided window classes introduced on page 3-13. As such, they are displayed for your application by OS/2, which also helps your application operate them. See Chapter 7, "Dialog Boxes and Controls" for details.

This chapter and the others in this part of the guide describe how you specify and use the following objects and resources that your application can use to create a Common-User-Access-conforming user interface:

- Standard window action bars and pull-downs
- Dialog boxes and controls
- Icons, pointers, and bit maps
- Fonts
- Text strings
- Resource script files.

Note: Some icons produced using these resources will appear visually different from those shown in the CUA specification, examples are:

Max/Min/Restore

Pushbuttons

Scroll bar arrows.

The applications provided with OS/2 1.2 generally conform to CUA but there may be minor deviations.

Limits for Number of Windows

The recommended maximum number of windows is:

Overall System Limit: 1200 windows

One Application: 100 windows

The above numbers are approximate, and depend on the resources used by each window. The numbers are reduced by extensive use of resources such as input queues, accelerator tables, and window words.

You should remember that the WC_FRAME window is a composite window that can contain several components, such as the title bar and action bar, which are actually child windows. Dialog boxes also consist of a frame window, plus child windows that you specify in the dialog template. Each button, list box, entry field, or other control that is defined in the dialog box is a separate window.

When creating windows, check for errors. If your application receives an error on any of the various calls that create windows, indicating that the window was not created, do one or more of the following:

- Free up some other windows and try again
- Put up a message box informing the user of the problem (this works only when the message box uses up fewer windows than the operation that failed)
- Issue a VioPopup call and put out a message using Vio API services.

In addition to the above, your program should be prepared to shutdown in a tidy and efficient manner at any time that the window limit is reached, without creating a window. Again, VioPopUp can be used to interact with the user if the window limit is reached during shutdown.

Chapter 4. The HELLO1 Sample Application

This chapter uses the HELLO1 sample application to demonstrate some concepts of Presentation Manager (PM) programming.

The HELLO1 sample application creates a window on the display screen, and displays some text in white characters on a red background, as shown in Figure 4-1.

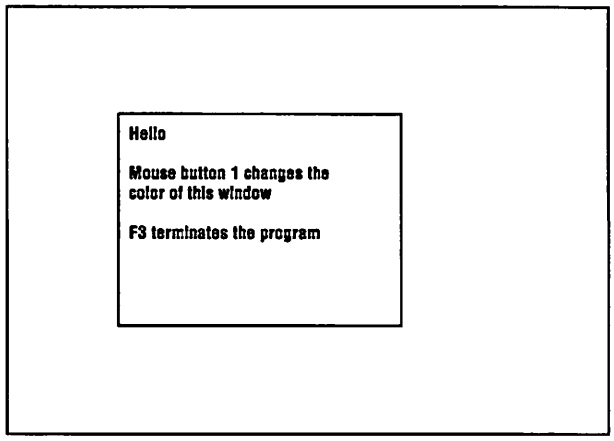


Figure 4-1. Appearance of Window Displayed by the HELLO1 Application

When the user positions the pointer in the window, and clicks mouse button 1 (typically the left button), the background color switches between red and blue. When the window is displayed, if another window is placed on top of it, and later removed, the window belonging to the example application redraws itself. The only keyboard input the application recognizes is the F3 key. When the user presses the F3 key, the application terminates.

The HELLO1 application demonstrates the following features of a PM application:

- The source code structure common to *all* PM applications (and as defined by SAA Presentation Interface).
- Message communication with OS/2, other PM applications, and the application.
- Responsiveness to the user.

The HELLO1 application and most of the other samples described in this book are written in the C/2 programming language. There are comments in the code, contained within /* and */ delimiters, and in text following the application examples. These explain what happens in the code, in terms that can be understood regardless of the language you use to write your application. Also, most of the application consists of OS/2 function calls. There is, therefore, very little C/2 code in the application. (In OS/2 1.2, PM applications can be written in C/2, Macro Assembler/2, COBOL/2, or FORTRAN/2.) However, it will help you to understand the samples if you have a basic working knowledge of the C/2 language syntax. Each OS/2 function call and its parameters are described in full in the *Control Program Programming Reference*.

There are no COBOL or FORTRAN versions of HELLO1. If such versions existed, they would be slightly different from the C/2 version, as described in "Nonreentrant Equivalent of HELLO1" on page 4-10. If you are programming in COBOL or FORTRAN, you should read through the description of HELLO1, as it will give you the context to understand the nonreentrant support.

Example Code: HELLO1 Sample PM Application

```
#define INCL_WININPUT                   /* Select part of header        */
#include <os2.h>                        /* PM header file               */

#define ID_WINDOW    255

/*****************************************************************************/
/* Function prototypes                                                       */
/*****************************************************************************/
MRESULT EXPENTRY MyWindowProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 );
void cdecl main( void );

                                        /* Define parameters by type    */
HAB   hab;                              /* PM anchor block handle       */
COLOR colorWnd = CLR_RED;               /* Window Color                 */

/*********************  Start of main procedure  ***********************/
void cdecl main(  )
{
  HMQ  hmq;                             /* Message queue handle         */
  HWND hwndClient;                      /* Client area window handle    */
  HWND hwndFrame;                       /* Frame window handle          */
  QMSG qmsg;                            /* Message from message queue   */
  ULONG flCreate;                       /* Window creation creation flags*/

  hab = WinInitialize( NULL );          /* Initialize PM                */
  hmq = WinCreateMsgQueue( hab, 0 );    /* Create a message queue       */

  WinRegisterClass(                     /* Register window class        */
     hab,                               /* Anchor block handle          */
     "MyWindow",                        /* Window class name            */
     MyWindowProc,                      /* Address of window procedure  */
     0L,                                /* No special Class Style       */
     0                                  /* No extra window words        */
     );

  flCreate = FCF_TASKLIST;              /* Set Frame Control Flag       */

  hwndFrame = WinCreateStdWindow(
              HWND_DESKTOP,             /* Desktop window is parent     */
              0L,                       /* No Class Style               */
              &flCreate,                /* Frame creation flag          */
              "MyWindow",               /* Client window class name     */
              "",                       /* No window text               */
              0L,                       /* No special class style       */
              NULL,                     /* Resource is in .EXE file     */
              ID_WINDOW,                /* Frame window identifier      */
              &hwndClient               /* Client window handle         */
              );

  WinSetWindowPos( hwndFrame,           /* Set the size and position of */
                   HWND_TOP,            /* the window before showing.   */
                   100, 100, 250, 200,
                   SWP_SIZE | SWP_MOVE | SWP_ACTIVATE | SWP_SHOW
                 );
```

```
/*****************************************************************/
/* Get and dispatch messages from the application message queue  */
/* until WinGetMsg returns FALSE, indicating a WM_QUIT message.  */
/*****************************************************************/
  while( WinGetMsg( hab, &qmsg, NULL, 0, 0 ) )
    WinDispatchMsg( hab, &qmsg );

  WinDestroyWindow( hwndFrame );        /* Tidy up...            */
  WinDestroyMsgQueue( hmq );            /* and                   */
  WinTerminate( hab );                  /* terminate the application */
}
/*********************  End of main procedure  ********************/

/*******************  Start of window procedure  ******************/
MRESULT EXPENTRY MyWindowProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
  HPS    hps;                           /* Presentation Space handle */
  RECTL  rc;                            /* Rectangle coordinates     */
  POINTL pt;                            /* String screen coordinates */

  switch( msg )
  {
    case WM_PAINT:
      /*************************************************************/
      /* Window contents are drawn here in WM_PAINT processing.    */
      /*************************************************************/
                                        /* Obtain a cached micro PS  */
      hps = WinBeginPaint( hwnd, NULL, &rc );
      WinFillRect( hps, &rc, colorWnd );/* Fill invalid rectangle    */
      pt.x = 10L; pt.y = 190L;          /* Set the text coordinates, */
      GpiSetColor( hps, CLR_WHITE );    /* Set color of the text     */
      GpiCharStringAt( hps, &pt, 5L, "Hello" );      /* Draw Text    */
      pt.y = 170L;
      GpiCharStringAt( hps, &pt, 26L, "Mouse button 1 changes the" );
      pt.y = 160L;
      GpiCharStringAt( hps, &pt, 20L, "color of this window" );
      pt.y = 140L;
      GpiCharStringAt( hps, &pt, 25L, "F3 terminates the program" );
      WinEndPaint( hps );                       /* Drawing is complete */
      break;
    case WM_BUTTON1DOWN:
      /*************************************************************/
      /* Mouse button 1 has been clicked, so first ensure we have the */
      /* input focus and then toggle the window color between RED and */
      /* BLUE and cause the window to be redrawn.                  */
      /*************************************************************/
      WinSetFocus( HWND_DESKTOP, hwnd );
      colorWnd = ( colorWnd == CLR_RED ) ? ( CLR_BLUE ) : (CLR_RED );
      WinInvalidateRect( hwnd, NULL, TRUE );
      break;
    case WM_CHAR:
      /*************************************************************/
      /* Character input is processed here.                        */
      /* The first two bytes of message parameter 2 contain        */
      /* the character code.                                       */
      /*************************************************************/
      if( SHORT2FROMMP( mp2 ) == VK_F3 )    /* If the key pressed is F3,*/
        WinPostMsg( hwnd, WM_QUIT, 0L, 0L );/* post a quit message to   */
      break;                                /* end the application.     */
    case WM_CLOSE:
```

```
      /*********************************************************************/
      /* This is the place to put your termination routines                */
      /*********************************************************************/
      WinPostMsg( hwnd, WM_QUIT, 0L, 0L );  /* Cause termination           */
      break;
    default:
      /*********************************************************************/
      /* Everything else comes here.  This call MUST exist                 */
      /* in your window procedure.                                         */
      /*********************************************************************/

      return WinDefWindowProc( hwnd, msg, mp1, mp2 );
  }
  return FALSE;
}
/*********************  End of window procedure  *************************/
```

HELLO1 Sample Application Walk-Through

Include Files

The statement `#include <os2.h>` in HELLO1 instructs the C/2 compiler preprocessor to include the file OS2.H as a header file to the application. The file OS2.H itself includes files containing declarations of the data structures and calls in the OS/2 API. Having include files in your application saves you from writing your own declarations for the OS/2 API calls covered by the include files. Because you are unlikely to ever use the entire OS/2 API, you can use #define statements with symbolic names of the form INCL_WININPUT, as in the example, to include only the parts of the include files that contain the required API subsets. The symbolic name of the include file section in which each call resides is given in the description of each call in the syntax manual for the programming language you use.

The statement `#define ID_WINDOW 255` assigns the value 255 to the symbolic identifier ID_WINDOW. In a real application, you would place all such statements in an application include file, and use a #include statement to include the contents of the file in your application.

Function Prototypes

The MyWindowProc statement is a function prototype; a predeclaration of the application window procedure MyWindowProc and its parameters. In C/2, it is strongly recommended that you provide function prototypes for all functions, so that the compiler can check the parameter types. MyWindowProc is actually called by OS/2, as explained later in this chapter. All application procedures that can be called by OS/2 must use the PASCAL calling convention for passing parameters. The EXPENTRY keyword is defined in one of the OS/2 include files as FAR PASCAL, which satisfies this convention. The one function called by OS/2 that does not need to use the keywords is the Main function.

The Main Function

The Main function starts with the statement

```
void cdecl main(  );
```

and is the entry point of a PM application. All windowed C/2 applications must have it.

The statements:

```
HAB   hab;
COLOR colorWnd = CLR_RED;
```

and

```
HMQ  hmq;
HWND hwndClient;
HWND hwndFrame;
QMSG qmsg;
ULONG flCreate;
```

define by data type the variables used in the Main function. The data types are in uppercase in the left column, and correspond to the standard data types defined in the include file. The standard data types are listed in the *Control Program Programming Reference*. The variables used by the application are in the right column. They typically have a lowercase prefix that is an abbreviation of the type of variable that they represent. For example, when you create certain objects under OS/2, the system returns a *handle* for the object. A handle is a unique number that OS/2 uses to refer to the object. You use the handle as an input parameter to calls that refer to the object. A variable holding a *h*andle used by a *wind*ow has the prefix *hwnd*.

The WinInitialize call (WMINIT) initializes the PM application programming environment. WinInitialize reserves an *anchor block*. An anchor block is an area of PM-internal resources that is allocated to a thread that calls WinInitialize. WinInitialize returns an anchor block handle, hab. An anchor block handle is a unique number, supplied by OS/2, used to identify the anchor block and distinguish it from the anchor blocks of other threads. The anchor block handle is required as an input parameter to some subsequent calls. The anchor block can contain error information, and be used for debugging an application, as described in Appendix A, "API Error Processing." Each thread of an application that uses the PM API should, therefore, contain a WinInitialize (WMINIT) call.

Creating a Message Queue

The WinCreateMsgQueue (WMCMQ) call creates the message queue on which OS/2 places messages. The application must do this to receive user input. WinCreateMsgQueue returns a message queue handle. This is a unique number, supplied by OS/2, that identifies the queue, and can be used by subsequent calls.

Each thread of an application that can receive PM messages must have its own message queue.

Registering a Window Class

The WinRegisterClass (WMRCL) call registers the window class and defines to OS/2 the window procedure and other information to be shared by all windows of the class being registered.

By default, WinRegisterClass (WMRCL) registers a *private window class*. A private window class can be used to create windows only by the application that registers it. In the example, WinRegisterClass gives the window class the unique class name MyWindow, which can be used as a parameter to a WinCreateWindow or WinCreateStdWindow call that subsequently creates a window of the registered class.

WinRegisterClass (WMRCL) also tells OS/2 the address of the application window procedure (MyWindowProc in the example) that processes messages input to windows of the MyWindow class. In the sample, MyWindowProc is a routine further on in the application. Alternatively, the window procedure for a private window class can be loaded into the application from a dynamic link library, using the DosLoadModule call. In this case, the application to resolves the address of the window procedure before using WinRegisterClass. See "Registering a Private Window Class" on page 13-9 for details.

The window class in the example does not use a particular *class style*.

The WinRegisterClass in the example does not allocate any window words. See "Using Window Words" on page 13-14 for an example.

Creating a Window

When the application has registered a window class, it creates a window of that class. WinCreateStdWindow (WICRTS) implicitly creates a window of the system-provided WC_FRAME class, known as a standard window, and also creates the application-registered window. Normally, such a window has a frame, containing various controls that are drawn and controlled by OS/2, and with which the user can interact. The frame encloses an area known as the client area. This area is reserved for the application-registered window, the client window. The client window is

created as a child of a frame window, which has the desktop window HWND_DESKTOP (the display screen) as its parent. The client window can have child windows of its own, if required. HELLO1 uses WinCreateStdWindow for the sake of simplicity, but is not the recommended method of creating a standard window. See Chapter 5, "The Standard Window" for a description of the recommended method.

The appearance of the frame and its controls is specified using the FCF_ frame creation flags. In HELLO1, a single FCF_ flag only is used. FCF_TASKLIST places the name of the program in the task list displayed by the Task List. The absence of any other FCF_FLAGS means that only the client window appears. The frame is invisible. See Chapter 5, "The Standard Window" for how to create a standard window with a frame and other controls.

The WS_ styles, and the FCF_ flags are listed in the *Control Program Programming Reference*.

WinCreateStdWindow (WICRTS) specifies the application-registered class to which the client window belongs; in the example, the class is MyWindow, as registered earlier using WinRegisterClass (WMRCL).

You can specify text for a window title as a parameter to WinCreateStdWindow (WICRTS). For simplicity, the example does not do this.

In the sixth parameter of WinCreateStdWindow, you can specify the WS_ window style for the client window. In the example, the client window has no particular window style.

Frame windows can load resources that are external to the application code, such as text strings, and icons. HELLO1 does not use resources.

WinCreateStdWindow (WICRTS) gives the created window an application-defined identifier ID_WINDOW, which has the value 255, set earlier in the application by a #define.

WinCreateStdWindow (WICRTS) creates a window, returns the frame window handle in hwndFrame, and gets the client window handle in hwndClient. A window handle is a unique number, provided by OS/2, that identifies a window. The window handle can subsequently be used as a parameter by calls that operate on the window, and as a parameter to messages passed to the window.

Positioning and Activating a Window

You can position and size a window using the call WinSetWindowPos (WISPOS). To ensure that a real application is device-independent, you can query the display device, and specify the size of the window to match the display screen and the amount of data displayed.

Every window on the screen, regardless of the application to which it belongs and whether it is visible, has a position in a viewing order. In the example, HWND_TOP brings the created window to the top in the viewing order.

The arguments 100, 100, 250, 200 are two pairs of pel coordinates that respectively specify the position and size of the window. The origin of the window (the lower-left corner) is positioned 100 pels to the right and 100 pels up from the origin of the desktop window (the lower-left corner of the screen). The created window is 250 pels wide, and 200 pels high. WinSetWindowPos (WISPOS) sets the position and size of the frame window and consequently of the client window (which is positioned and sized by default inside its parent frame window). WinSetWindowPos (WISPOS) uses the handle of the frame window as a parameter. The SWP_SIZE and SWP_MOVE options set the size and position of the frame, from the coordinates 100, 100, 250, 200. The SWP_SHOW option shows the window (makes it visible). The SWP_ACTIVATE option makes the window active. The active window is the application main window that is (or contains the window that is) the input focus window. The input focus window is the window that receives input from an input device (typically the keyboard). There can be only one active window in the whole system at any time. The SWP_ACTIVATE option causes the WM_ACTIVATE message to be sent to the newly activated window, to tell it that the user has chosen it to be active. Activating a window does not affect the appearance of the window in HELLO1. HELLO1 does not process the WM_ACTIVATE message in its window procedure, so it passes the message through to the default window procedure instead.

If a main window of another application was active before the example application called WinSetWindowPos (WISPOS), WM_ACTIVATE would also be sent to that window, telling the window it is no longer the active window. The WM_ACTIVATE message can also be sent by the

WinSetActiveWindow (WMSACT) call. The message contains the last parameter of WinSetActiveWindow, TRUE, to indicate that the active window should set itself or one of its windows to be the input focus window. Because WinSetWindowPos with the SWP_ACTIVATE option and WinSetActiveWindow can affect other applications in this way, they should be called only as a result of user input.

WinSetWindowPos, and WinSetActiveWindow are examples of calls that *implicitly* send a message. Compare this with the calls WinDispatchMsg (WMDISP) and WinSendMsg (WMSEND), that explicitly send messages. Sending a message to a window calls the window procedure for that window. Window procedures can, therefore, be recursive, as explained in "Window Procedure Recursion" on page 4-9.

The Message Loop

The message loop is the heart of the application, and uses the calls WinGetMsg (WMGET) and WinDispatchMsg (WMDISP). WinGetMsg removes messages one by one from the application queue. As long as WinGetMsg returns a value of TRUE, WinDispatchMsg keeps sending messages to the window. WinDispatchMsg effectively calls the window procedure for the window defined in the qmsg structure. In the example, this can only be the application's window procedure. However, more complex applications can have several windows, and so can have several window procedures.

If the window is sent a WM_QUIT message, WinGetMsg returns a value of FALSE, and the application breaks out of the loop. (Later, in the window procedure, in response to the user pressing the F3 key, the application sends the window the WM_QUIT message.)

Application Termination

WinDestroyWindow (WIDEL) removes a window from the screen, and deletes the definition of the window held by OS/2. If the window has any child windows (for example, the client window in the example), they are also removed from the screen and deleted from the system.

WinDestroyMsgQueue (WMDMQ) destroys the application's input queue. When this is done, the application can no longer receive input.

WinTerminate (WMTERM) ends the application's use of the Win and Gpi calls. A PM application would, in general, terminate at this point, because you have removed the means of displaying the results.

The Window Procedure

The window procedure defines the functions of a window class. All the windows of a particular class use the window procedure for that class. In the example the MyWindowProc window procedure has four parameters, which are the elements of the message being passed to it:

- The handle of the window to which the message is addressed (a 16-bit word)
- The message type (a 16-bit word)
- First message parameter (a 32-bit double-word)
- Second message parameter (a 32-bit double-word).

The message type is defined in OS/2 as a two-character prefix, indicating the type of window that is the source of the message, an underscore character, and a string of characters that describes the purpose of the message. For example, the message type WM_PAINT has a prefix of WM because it is a Window Message. PAINT indicates to the target window that it may want to repaint itself.

The message parameters contain data values that convey information to the window procedure. Their content varies according to the message type. Each PM message is described in full in the *Programming Reference: Volume 2.*

The message types processed by the window procedure in the example application are described in the following sections, in the order that they are *received* by the window procedure. The order in which you place your case statements is not significant, however. In fact, you should not expect any message to form part of a sequence or to arrive at a particular time. *Your application should process each message in its own right.*

HELLO1 processes four messages: WM_PAINT, WM_BUTTON1DOWN, WM_CHAR, and WM_CLOSE. After each message is processed, the window procedure returns FALSE (0).

The WM_PAINT Message

This message notifies the application that it might want to repaint the contents of the window. The message occurs when a window is created, and when it needs repainting. The WM_PAINT processing is where all the drawing relating to a window takes place, including the initial drawing.

WM_PAINT can also occur if a window that overlays the application's window is removed. In this case, WM_PAINT informs your application so that it can perform any redrawing if required.

How a window draws itself and what it draws differ, depending on the window and the application. The method in the example application is a simple technique that can be used by windows with a small amount of simple data to draw.

The first action an application must do when drawing in a window is to create a *presentation space*. The application begins painting using WinBeginPaint (WIBPNT). The presentation space handle is in hps. The handle subsequently is used as a parameter on most graphics calls. No presentation space has yet been created, so WinBeginPaint specifies NULL in the second parameter. The WinBeginPaint (WIBPNT) gets a micro PS from a cache of micro PSs kept by OS/2. This is the implicit form of presentation space creation. (A cache micro PS can also be initially obtained using WinGetPS (WPGPS) or WinGetScreenPS (WPGSPS).) A micro PS is contained inside the window boundary.

In its final parameter, WinBeginPaint gets the address of a data structure containing the coordinates of the rectangle that needs repainting. A more sophisticated application can optimize repainting by using the returned rectangle to only repaint the part of the presentation space that needs repainting.

The coordinates returned by WinBeginPaint (WIBPNT) are for the entire presentation space, and are used by WinFillRect (WIFRCT), which fills a rectangle of this size with the color colorWnd, defined as constant CLR_RED earlier in the program.

GpiSetColor (GSCOL) sets the color of the graphics text to white. GpiCharStringAt (GICHAR) draws the various character strings containing user instructions at the position given in the x and y coordinates.

Finally, WinEndPaint (WIEPNT) ends the drawing operation by destroying the presentation space, and indicating to OS/2 that the window is updated. See "Part 4. Presentation Manager Applications – Using the Graphics Programming Interface" for a description of the Gpi calls in the Graphics Programming Interface.

The WM_BUTTON1DOWN Message

This message indicates that the user has clicked button 1 on the mouse with the pointer over part of the visible area of the window. WinSetFocus (WMSFOC) gives the window the input focus. The window initially has the input focus, but the call is needed in case the user switches to another application and then back to the sample.

The example application switches the background color between red and blue, on receipt of the WM_BUTTON1DOWN message. The WinInvalidateRect (WINVR) call can be used to specify the coordinates of the update region of a window that needs repainting. In the example, a value of NULL in the second parameter specifies that the whole window needs repainting.

Although the sample program processes the WM_BUTTON1DOWN message, it is more usual to process the WM_BUTTON1UP in a real application.

The WM_CHAR Message

This message indicates that the user has pressed or released a key on the keyboard. This covers every key on the keyboard, including alphanumeric keys, function keys, and shift keys. Information in the WM_CHAR message helps the application interpret keyboard input from the user. There are two basic pieces of information:

- A character code, which is the ASCII value associated with an alphanumeric key. This is typically zero for function keys.
- A virtual key code (VK_* value), which identifies that a particular function key has been pressed. This value is zero for alphanumeric keys.

Some information relates to the current shift status of the keyboard, for normal shift, Alt shift, and

Ctrl shift. A flag indicates whether the key was pressed or released. A count value indicates how many times typematic keys, held down by the user, have been repeated.

In the example, the WM_CHAR processing tests only whether the user presses the F3 key. The application does this by using the SHORT2FROMMP macro to extract the second short value from the second message parameter, and test whether it contains the VK_F3 value (the value of the constant returned by OS/2 when the user presses F3). If it does, WinPostMsg (WMPOST) posts the WM_QUIT message to the application, to terminate it. This causes the WinGetMsg (WMGET) function to return FALSE and the application leaves the message processing loop and drops through into the termination code, as explained in the section of the main procedure, described in "Application Termination" on page 4-7.

Posting the WM_QUIT message is the usual method of terminating an application.

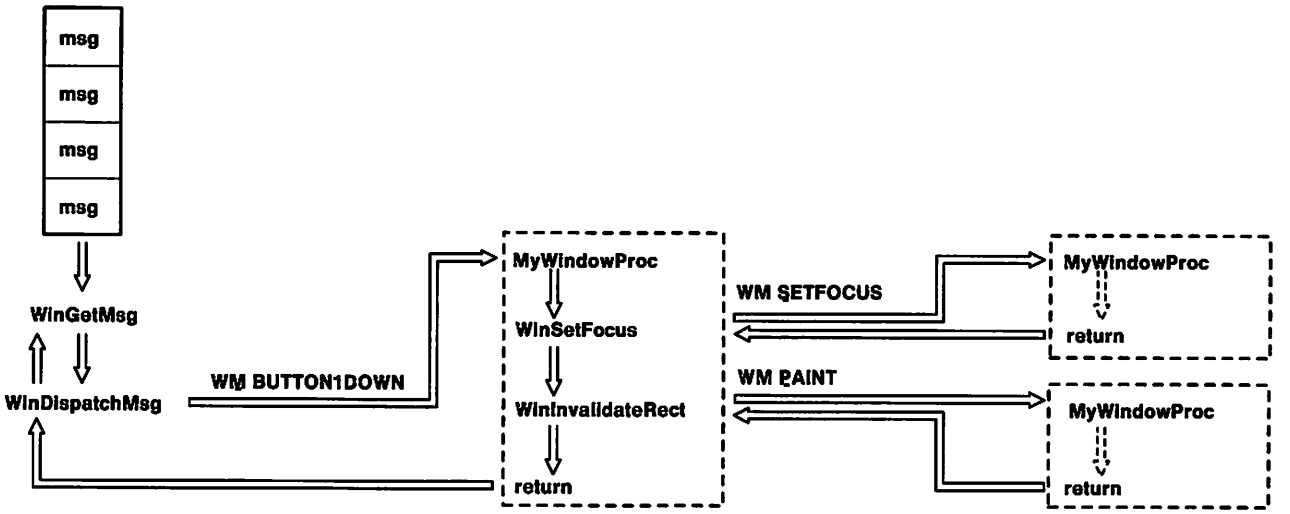


Figure 4-2. Window Procedure Recursion in HELLO1

The WM_CLOSE Message

When the user closes the program from the system menu, rather than from the program, OS/2 sends the application a WM_CLOSE. The application deals with it by posting the WM_QUIT message.

Default Processing of Messages

The default case at the end of the window procedure handles all types of message not explicitly processed by the window procedure. The processing performed by the example application is typical of *all* window procedures.

WinDefWindowProc (WMDWP) calls a system-provided default window procedure, passing on the message received, but not processed by, the application window procedure. The return value from the default window procedure is used as the return value for the application window procedure. This value must be returned.

Using the default window procedure, an application can process *any* message it is sent. This is a requirement for every window procedure. The default window procedure gives a default response to every kind of message, some NULL (no processing done), and some real responses. The default window procedure action for each message is documented in the *Control Program Programming Reference* manual.

Window Procedure Recursion

The flow of messages through the window procedure in HELLO1 shows that window procedures can be recursive, as shown in Figure 4-2.

When the user clicks mouse button 1, a WM_BUTTON1DOWN message arrives on the application input queue. This is obtained by WinGetMsg (WMGET), and sent to the application window procedure by WinDispatchMsg

(WMDISP). WM_BUTTON1DOWN arrives at its case statement in the application window procedure, and is processed by calling WinSetFocus (WMSFOC).

WinSetFocus sends a WM_SETFOCUS message to the application window procedure. There is no case statement for the message, so it drops through to the default case and is processed by the default window procedure. This returns, so that calling the application window procedure by WinSetFocus is complete.

The processing of the WM_BUTTON1DOWN message continues with a call to WinInvalidateRect (WINVR) which sends a WM_PAINT message to the application window procedure. The window repaints itself with the new background color, and the window procedure returns. Calling the application window procedure by WinInvalidateRect is complete. The application window procedure returns and control returns to WinDispatchMsg.

The application window procedure is called three times as a result of the application receiving the WM_BUTTON1DOWN message. The first time it is called by WinDispatchMsg (WMDISP). In processing the WM_BUTTON1DOWN message, the application window procedure is called twice more; once by WinSetFocus (WMSFOC), and once by WinInvalidateRect (WINVR).

The processing of a message can call a function that itself sends a message to the application window procedure. Processing *that* message can call a function that itself sends a message, and so on. The extent of recursion in an application window procedure can, therefore, be far greater than in HELLO1. A calling stack of uncompleted calls and unprocessed messages can result. You must, therefore, always ensure that the processing of a message by calling a window procedure is resolved so that the calling stack is cleared. Otherwise, your program can run out of stack space and get a system error. You should take particular care where the processing of messages involves calling another thread.

Recursion and Local Variables

Because window procedures can be called recursively, you should keep the number of local variables to a minimum, to prevent overflow of the stack. You can do this by processing individual messages in separate routines that are called from the window procedure. You can declare any local variables that you need in the separate routines. This way, space on the stack for local variables is allocated only when needed.

Nonreentrant Equivalent of HELLO1

A nonreentrant equivalent of HELLO1 would have a similar structure. The main difference, apart from the call names being the short versions, would be that a nonreentrant version would itself be able to process only posted messages. (The language-support window procedure processes sent messages.) It would, therefore, not have a proper window procedure, but a window routine. Examples of differences are described in the following sections.

Include Files

Instead of OS2.H, COBOL/2 applications COPY a file containing the required constants from OS2.CIN and PM.CIN. FORTRAN/2 applications INCLUDE an equivalent file. You must also COPY or INCLUDE the header files for any structures used by PM calls in your application.

You can use similar statements for your application header files.

Function Prototypes

These are not possible in COBOL or FORTRAN.

The Main Function

This would be a section in the procedure division of a COBOL/2 program, or a routine in a FORTRAN/2 program. Unlike the C/2 program, the WMRCL call would register a NULL window class, to indicate that the application cannot provide a window procedure.

The message processing loop would consist of a WMGET and a WMDISP. The two calls would be separated by an IF statement that tested whether the target of the message is our application window. If it is, the application calls a window routine. In COBOL/2, this is usually a PERFORM statement. In FORTRAN/2, it is a CALL to a routine.

The Window Routine

The window routine would consist of a series of IF statements, each testing for the same messages as the window procedure in the C/2 version. The IF statement for the WM-PAINT message would also test for the WM-PPAINT message as posted back by the language-support window procedure.

In the processing of the WM-BUTTON1DOWN message, there are WinSetFocus (WMSFOC) and WinInvalidateRect (WINVR) calls. "Window Procedure Recursion" on page 4-9 describes how, in a reentrant application, these calls send WM-SETFOCUS and WM-PAINT messages to the application window, causing its window procedure to be called to process them. In a nonreentrant application, the sent messages are intercepted by the language-support window procedure, and posted to the application queue as WM-PSETFOCUS and WM-PPAINT messages. The window routine does, therefore, receive the messages, but as posted-on versions. Also, the WinInvalidateRect (WINVR) call in the C/2 sample is an example of a call to which, in C/2, you can supply a NULL pointer. On WinInvalidateRect the NULL pointer is in the second parameter and specifies the whole window as the rectangle to be added to the window's update region. In COBOL/2 and FORTRAN/2, there is no equivalent to the NULL pointer. Instead, you explicitly supply the coordinates of the rectangle, or use the WinInvalidateRegion (WINVRG) call, to which you can supply a NULL handle. NULL handles are supported in COBOL/2 and FORTRAN/2, and specify the the whole window as a region to be added to the update region.

In the processing of the WM-CHAR message, the SHORT2FROMMP macro would not be used to extract the character code from the message parameter, as the macro is supported only in C/2 and Macro Assembler/2. Instead, the WUQVAL (WinQueryValue) call would be used.

The additional processing needed in COBOL/2 and FORTRAN/2 is described in Chapter 11, "The DLG1 Nonreentrant Sample Application."

Traditional Equivalent of HELLO1

Here is a pseudocode version of a traditional equivalent of the HELLO1 application:

```
Begin

  Initialize System

  Draw Red Background to Screen

  Draw Text Strings ("Hello")
   .
   .

  Repeat

    If Mouse Clicked, Change Background Color

    Char = GetNextKeystroke

  Until (Char = F3)

  Terminate System

End
```

The above application is in pseudocode as only its structure is important, not its programming details.

Clearly the above application is simpler than the HELLO1 application. It consists of a single procedure (assuming that some system services are provided for the drawing functions and the character input). It is also much shorter than the example application. However, the simplicity is deceptive. The "traditional" application is functionally deficient in a number of important ways:

- The application goes through only once.
- There is no capability to redraw the screen.
- No allowance is made for any other applications in the system.
- There is no capability for the application to put itself on top to receive input.
- There is no mouse input handling.

These deficiencies can be solved, but only at the expense of additional complexity. They also require a more complex supporting system, particularly for coordinating mouse and keyboard input. However, the most serious failing of the "traditional" application is the difficulty in extending the user interface. The ease of extending the user interface of PM applications is shown in the Chapter 6, "The HELLO2 Sample Application."

The HELLO1 sample application is very basic, particularly in its user interface. It has little function, and does not make its function visible.

However, a few simple changes can give the example application a more sophisticated user interface. In particular, the user interface can be extended to display an SAA Common-User-Access-conforming standard window, as described in the next chapter, and as coded in Chapter 6, "The HELLO2 Sample Application."

Chapter 5. The Standard Window

The *standard window* is a standard layout for a window. In its full form, the appearance of a standard window is as shown in Figure 5-1.

An instance of a WC_FRAME class window encloses the client area, which can contain an instance of a window class implemented by your application (the client window). Bounding the client window, a WC_FRAME window can optionally include several standard interface control windows known as frame controls. All the components of the standard window are in fact separate windows, which are child windows of the frame window. Each frame control is an instance of one of the system-provided WC_ window classes, and can be separately included through the use of a corresponding FCF_ frame creation flag, when creating a standard window using WinCreateStdWindow (WICRT) or WinCreateWindow (WICRTS). There is also a composite frame creation flag FCF_STANDARD that is equivalent to several of the frame creation flags, and therefore includes several frame controls simultaneously. All the FCF_ flags are listed in the *Programming Reference: Volume 2.*

Several of the windows that can comprise the standard window have a unique identifier value, with an FID_ prefix, used to identify the window:

FID_TITLEBAR
Title bar

FID_MINMAX
Maximize/minimize/restore buttons

FID_MENU
Action bar and associated pull-down menus

FID_SYSMENU
System menu icon

FID_HORZSCROLL
Horizontal scroll bar

FID_VERTSCROLL
Vertical scroll bar

FID_CLIENT
Client area containing the client window, where the application draws its data.

The frame window arranges the frame controls and the FID_CLIENT window when the frame window is sized or moved. It also routes specific messages to its frame controls and the FID_CLIENT window.

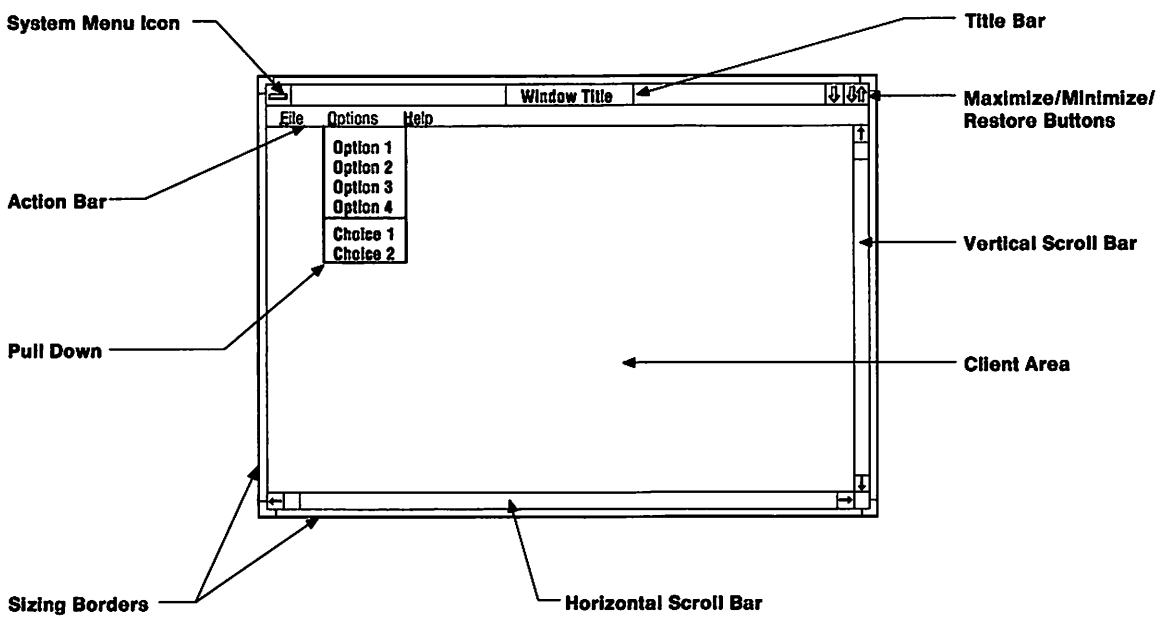


Figure 5-1. The Standard Window

The standard window is a good example of the structuring and separation of data. The application's data is displayed in the client window, and the menu windows contain the functions relating to the data in the client window. The title bar holds the name of the window and lets the user move the window. The sizing borders let the user change the size of the window.

Window Ownership

A window can *own* another window. The WC_FRAME window is the *owner* of its frame control children. This means that when an event that could be of interest to the frame happens to the control, the frame control sends or posts a *notification message* to its owner. Examples of events that could be of interest to the frame are, say, the user clicking the mouse in a scroll bar, or selecting a choice in a pull-down. As a result of these events, the frame may want to take some action, and is therefore notified.

When an owner window wants to affect or query the state of one of its control windows, it sends a *control message* to it.

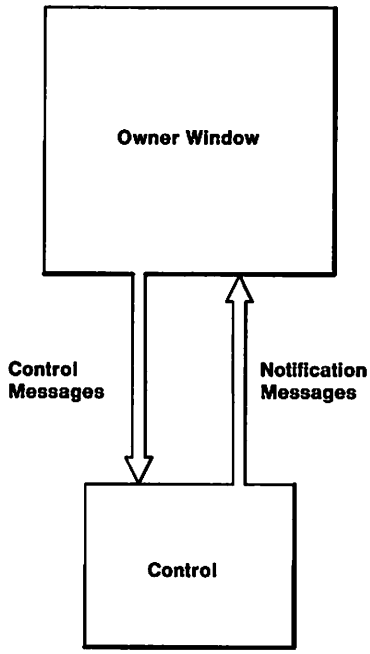


Figure 5-2. Message Communication Between Owner and Owned Windows

For example, when a user interacts with part of a vertical scroll bar in a standard window, the scroll bar sends the frame window a WM_VSCROLL notification message with an SB_ command value. The command value indicates the nature of the user's interaction. If the frame or application window needs to control the status of a scroll bar, it can send it any one of several control messages that have an SBM_ prefix, where SBM stands for scroll bar message.

There are other system-provided owner/owned relationships in the system. The notification and control messages vary according to the type of the control. Some of the messages are described in this chapter, and some in Chapter 7, "Dialog Boxes and Controls." They are listed for each control type in the *Control Program Programming Reference.*

There is usually no ownership relationship between the frame and the FID_CLIENT window. However, similarly to the relationship between the control windows and the frame, if an event that could be of interest to the client window happens to the frame, the frame window sends or posts a message to the FID_CLIENT window. For example, the frame window passes WM_VSCROLL messages on to the FID_CLIENT window.

A window can own any other window, but only when both windows belong to the same thread. It is not possible for a window in one thread to own a window in another thread.

Sending and Posting Messages to a Window

In the same way that the frame sends control messages to its control windows, your application can send control messages to the windows to control their status. The process is the same for sending or posting a message to any window.

When you want to query or affect the status of a window immediately, you can use WinSendMsg (WMSEND) to send a message. WinSendMsg bypasses the message queue associated with the receiving window, and sends the message directly to the window procedure of the receiving window. Control does not return to the calling application until the receiving window procedure has processed the message.

When you want to affect the status of a window, do not require a return value, and are content to let the receiving window process the message in its own time, you can use WinPostMsg (WMPOST) to post a message. WinPostMsg causes the message to be placed in the message queue associated with the receiving window. Control returns immediately to the calling application. Any processing done by the receiving window procedure does not happen until it receives the message from WinDispatchMsg (WMDISP).

As a parameter to the call that sends or posts a message to a window, you supply the window handle of the receiving window. Given a specified identifier of a window, the WinWindowFromID (WIQHID) call returns its window handle. You can use WinWindowFromID together with WinSendMsg or WinPostMsg to pass a message to any of the FID_ prefixed windows that comprise the standard window.

Creating a Standard Window

There are two ways of creating a standard window, containing an FID_CLIENT window. You can use WinCreateStdWindow (WICRTS), which creates a WC_FRAME window, and also creates an application-registered FID_CLIENT window for you. WinCreateStdWindow is convenient for simple situations, and is used in many of the programming tool samples. However, WinCreateStdWindow does not give you the maximum flexibility in passing information to a newly-created FID_CLIENT window. For example, you cannot set window ownership and window identifiers, or change the position of the FID_CLIENT window. If you want this sort of flexibility, then instead of using WinCreateStdWindow (WICRTS), you can use WinCreateWindow (WICRT) to create a WC_FRAME class window, and a further WinCreateWindow to create the FID_CLIENT window as a child of the frame.

WinDestroyWindow (WIDEL) destroys a window, its child windows, and its owned windows.

Example Code: Creating a Standard Window, Containing a Client Window

The following code, taken from the TEMPINIT part of the TEMPLATE sample application, is the recommended way of creating a standard window:

```
/***************************** Private Function *****************************/
/*                                                                          */
/* BOOL FAR TemplateInitApp( )                                              */
/*                                                                          */
/* Effects:  Registers Application Window Class                             */
/*           Creates Standard Window                                        */
/*                                                                          */
/****************************************************************************/

BOOL FAR TemplateInitApp( VOID )
{
    char szClassName[10];
    FRAMECDATA fcdata;
    HELPINIT   hinit;
    /************************************************************************/
    /* Load 'szClassName' from resource                                     */
    /************************************************************************/
    WinLoadString( hab, NULL, IDSNAME, sizeof(szClassName), (PSZ)szClassName );

    /************************************************************************/
    /* Register client window class                                         */
    /************************************************************************/
```

```
if (!WinRegisterClass( hab,
                       szClassName,
                       (PFNWP)TemplateWndProc,
                       CS_SIZEREDRAW,
                       0))
    return(FALSE);

/*************************************************************************/
/* fcdata contains control data structure for frame windows              */
/*************************************************************************/
fcdata.cb            = sizeof(FRAMECDATA);   /* Specify size             */
fcdata.flCreateFlags = FCF_STANDARD;         /* Standard frame           */
fcdata.hmodResources = NULL;                 /* Resource file not in DLL */
fcdata.idResources   = ID_TEMPLATE;          /* Resource file identifier */

/*************************************************************************/
/* Create frame window                                                   */
/*************************************************************************/
hwndFrame = WinCreateWindow(HWND_DESKTOP,       /* Desktop is parent     */
                            WC_FRAME,           /* Frame window class    */
                            (PSZ)NULL,          /* No window text or data */
                            0L,                 /* No window style       */
                            0,0,0,0,            /* Position and size     */
                            NULL,               /* No owner              */
                            HWND_TOP,           /* On top of siblings    */
                            ID_TEMPLATE,        /* Frame window identifier */
                            &fcdata,            /* Control data          */
                            NULL);              /* Reserved field        */

/*************************************************************************/
/* Create client window                                                  */
/*************************************************************************/
hwndClient = WinCreateWindow(hwndFrame,         /* Frame is parent       */
                             szClassName,       /* Class name            */
                             (PSZ)NULL,         /* No window text or data */
                             0l,                /* No window style       */
                             0,0,0,0,           /* Position and size     */
                             NULL,              /* No owner              */
                             HWND_TOP,          /* On top of siblings    */
                             FID_CLIENT,        /* Client window identifier*/
                             NULL,              /* Control data          */
                             NULL);             /* Reserved field        */

/*************************************************************************/
/* Set window visibility state ( Visible )                               */
/*************************************************************************/
WinShowWindow(hwndFrame, TRUE);

/*************************************************************************/
/* Get the window title given to the window by the system                */
/*************************************************************************/
WinQueryWindowText(hwndFrame, CCHMAXSTRING, szWindowTitle);

/*************************************************************************/
/* Initialize the Help Manager initialization structure, create an       */
/* instance of the Help Manager, and associate that instance with the    */
/* frame.                                                                */
/*************************************************************************/
hinit.cBytes                  = sizeof(hinit);
hinit.ulReturnCode            = NULL;
```

```
    hinit.pszTutorialName          = NULL;
    hinit.phtHelpTable             = (PHELPTABLE) (0xFFFF0000 | ID_TEMPLATE);
    hinit.hmodHelpTableModule      = (HMODULE)NULL;
    hinit.hmodAccelActionBarModule = (HMODULE)NULL;
    hinit.idAccelTable             = NULL;
    hinit.idActionBar              = NULL;
    hinit.pszHelpWindowTitle       = szWindowTitle;
    hinit.usShowPanelId            = CMIC_HIDE_PANEL_ID;
    hinit.pszHelpLibraryName       = "C:\\OS2\\HELP\\TEMPLATE.HLP";
    hwndHelp = WinCreateHelpInstance(hab, &hinit);
    if (hwndHelp != NULL)
    {
        if (!WinAssociateHelpInstance(hwndHelp, hwndFrame) )
        {
            WinDestroyHelpInstance(hwndHelp);
            hwndhelp = NULL;
        }
    }

    /*************************************************************************/
    /* Load various other strings                                            */
    /*************************************************************************/
    WinLoadString(hab, NULL, IDSTEXTSTR1, CCHMAXSTRING, szText);
    WinLoadString(hab, NULL, IDSTEXTSTR2, CCHMAXSTRING, szExampleText);

    return(TRUE);
}
```

The first WinCreateWindow (WICRT) call in the example creates the WC_FRAME window, and returns the window handle in hwndFrame. The second WinCreateWindow call creates the FID_CLIENT window, and returns the window handle in hwndClient.

In each WinCreateWindow call, the first parameter specifies the parent of the window being created. The WC_FRAME window is a child of the desktop. The FID_CLIENT window is a child of the WC_FRAME window. You can use WinCreateWindow (WICRT) to create a window of any system-provided or application-registered window class.

The second parameter specifies the window class. The frame window class is WC_FRAME. WinCreateWindow can be used in this way to create an instance of any of the system-provided window classes that have the prefix WC_. The FID_CLIENT window class is szClassName, as registered by WinRegisterClass.

The third parameter can contain the window text, or other class-specific data. For a WC_FRAME window, this is the text that is displayed in the title bar. The example does not specify any window text.

The fourth parameter can contain any of the system-provided WS_ window styles.

The fifth and sixth parameters can contain the x and y coordinates of the window position relative to its parent. The seventh and eighth parameters can contain the width and height of the window, in window coordinates. When the FCF_SHELLPOSITION or FCF_STANDARD flags are specified (see "Frame Window Control Data" on page 5-6), the window is created with a size and position determined by the user shell, rather than by the application.
If FCF_SHELLPOSITION is not specified and these values are zero, the window is created with width and height set to zero. Your program would then have to call WinSetWindowPos (WISPOS) to set the size and position.

The ninth parameter can contain the owner window handle.

The tenth parameter specifies the position of the window in the viewing order, relative to its siblings. HWND_TOP specifies that the window is first in the sibling viewing order, so it is displayed on top of its siblings.

The eleventh parameter specifies the window identifier. An application can use the identifier, instead of the window handle, to identify the window.

The twelfth parameter and thirteenth parameters are respectively the control data and presentation parameters, which are described in the next sections.

Passing Control Data and Presentation Parameters to a Newly-Created Window

Control data contains window data, the structure and content of which depend on the window class. The system-provided windows have control data with system-defined structures. A reentrant application can define whatever structure it requires for the control data for instances of application-registered windows.

Presentation parameters also contain class-specific data. The system-provided windows have presentation parameters containing attribute data. See "Using Presentation Parameters" on page 13-2 for details.

In the WM_CREATE message, the first parameter contains a pointer to the control data. The second parameter contains a pointer to a create structure, which itself contains the presentation parameters structure.

OS/2 sends a WM_CREATE to the window procedure of the window being created. The window procedure can use the information to initialize the window.

Nonreentrant applications do not receive the WM_CREATE message, as it is intercepted by the language-support procedure and processed in a default manner.

Frame Window Control Data

The WC_FRAME control data has the structure FRAMECDATA. FRAMECDATA looks like this:

```
typedef struct _FRAMECDATA {
    USHORT cb;
    ULONG flCreateFlags;
    HMODULE hmodResources;
    USHORT idResources;
} FRAMECDATA;
```

In the TEMPLATE example, fcdata has the structure FRAMECDATA. The flCreateFlags field can be set to any of the FCF_ values. The hmodResources field can be used to specify a module handle when the application's resource file is in a dynamic link library module. The example specifies NULL, indicating that the resource file containing the resources used by the frame window are in the compiled application. (You do not need to know how they get there for the moment. But "Writing a Resource Script File" on page 8-5 describes the process, if you cannot wait.) The idResources field specifies the identifier ID_TEMPLATE, that is used in the resource file to identify the resources used by the frame window.

Frame Window Resources

The resources used by a frame window are as follows:

Icon
: Displayed when the frame is minimized. The FCF_ICON flag associates the icon resource with the window, as does FCF_STANDARD.

Menu
: Supplies the contents of the action bar choices and their associated pull-downs. The FCF_MENU flag associates the menu resource with the window, as does FCF_STANDARD.

Accelerators
: Supplies the accelerators. The FCF_ACCELTABLE flag associates the accelerator table with the window, as does FCF_STANDARD.

Example Code: Frame Window Resources

In the following example code there are statements that define the frame window resources for the TEMPLATE application. You will notice the identifier ID_TEMPLATE is used to identify them, which is the resource identifier used in the application. The sections that follow describe the visual components of the standard window, and show how the components use the frame resources.

```
ICON ID_TEMPLATE   template.ico

MENU ID_TEMPLATE   PRELOAD
BEGIN
    SUBMENU  "~Example",                          IDMEXAMPLE
    BEGIN
        MENUITEM "~Dialog Example...\tCtrl+D",    IDMDIALOG
        MENUITEM SEPARATOR
        MENUITEM "E~xit" \tF3,                    IDMEXIT
    END
    SUBMENU    "~Help",                           IDMHELP
    BEGIN
        MENUITEM     "~Help for help...",         IDMHOWTO
        MENUITEM     "~Extended help...",         SC_HELPEXTENDED, MIS_SYSCOMMAND
        MENUITEM     "~Keys help...",             SC_HELPKEYS, MIS_SYSCOMMAND
        MENUITEM     "Help ~index...",            SC_HELPINDEX, MIS_SYSCOMMAND
        MENUITEM     SEPARATOR
        MENUITEM     "~About...",                 IDMABOUT
    END
END

ACCELTABLE      ID_TEMPLATE
BEGIN
    VK_F3,      IDMEXIT,   VIRTUALKEY
    "^D",       IDMDIALOG
END
```

Standard Window Visual Components

The components of the standard window present the user with a set of functions, as described in the following subsections.

Title Bar

You obtain the title bar using the FCF_TITLEBAR flag (or FCF_STANDARD). When the user presses either button on the mouse and holds it down while the pointer is over the title bar, the outline of the window is shown with a rectangular shadow box. The shadow box follows the mouse movements, indicating the position of the window. The window moves when the mouse button is released. The shadow box disappears and the window and all its contents are moved to the new position.

If the class style of the window is CS_MOVENOTIFY, a WM_MOVE message is sent to it to inform it that the move has occurred.

The keyboard user can move the window in a similar way, starting the process through a choice on the system menu, and using the keyboard arrow keys to move the window as required.

The title bar displays some text that is used to identify the window to the user. This is the window title of the standard window, which is set initially using the WinCreateWindow (WICRT) or WinCreateStdWindow (WICRTS) calls. The text subsequently can be changed by sending a WM_SETWINDOWPARAMS message to the window.

The window title should be given a name meaningful to the user. The name should include the name of any data on which it is operating. It is also conventional to add the program title to each window to indicate the program that owns the frame window. The FCF_TASKLIST flag does this for you. The resulting string in the window title is also sent to the Task List for entry on the task list. You can use the WinSetWindowText (WISTXT) to change the title at any time. The flag also updates the task list when the title changes, and removes the window title from the task list when the window is destroyed.

Maximize/Minimize/Restore Buttons

You obtain the maximize/minimize/restore buttons using the FCF_MINMAX, FCF_MINBUTTON, or FCF_MAXBUTTON flags (or FCF_STANDARD). The buttons are displayed at the right end of the title bar. The buttons are used to change the size of the window to its maximized or minimized state, or to restore the window to its normal state. The maximum size is determined by the application in response to the WM_QUERYMINMAXINFO message sent by the frame window to the FID_CLIENT window.

When the window is minimized, a small icon is displayed in place of the window itself. This icon is usually defined in a resource file and is automatically loaded when WinCreateWindow (WICRT) or WinCreateStdWindow (WICRTS) are called. You obtain the icon by specifying the FCF_ICON flag (or FCF_STANDARD).

The user causes the changes in state by using the mouse to select one of the icons. The keyboard user can do the same by using the system menu, which has appropriate choices available. When the size is changed in this way, a WM_MINMAX message is sent to the FID_CLIENT window to indicate that the change has been made.

Action Bar and Associated Pull-Downs

The choices in the action bar and the pull-downs associated with the action bar choices display the main functions of the window. Each action bar choice leads into a pull-down that is displayed when the corresponding action bar choice is selected. The pull-down consists of a vertical list of choices related to the action bar choice, and corresponding to individual functions of your application. The functions could be simple actions (for example, saving a file) or preferences (for example, choosing a color from a list). Choices in pull-downs are usually text, but they can be graphics.

You obtain the child windows to hold the action bar and pull-downs by using the FCF_MENU flag (or FCF_STANDARD). When created as part of a standard window, the action bar and pull-downs as a whole are known to the frame window by the identifier FID_MENU.

Action bar and pull-down choices are defined in a resource script file, in a simple text format. The resource script file and any resources to which it refers are compiled by a resource compiler into a resource file. The resource file is accessed by the WinCreateWindow (WICRT) and WinCreateStdWindow (WICRTS) functions, which use the identifier parameter to load the resource with the matching identifier value from the resource file. The action bar and pull-downs are created and displayed automatically, without further application involvement.

Action bars and pull-downs are designed for simple selection actions only. There are no complex forms of input such as entry fields or scroll bars. If such types of interaction are required, they should be placed in dialog boxes that are displayed when a choice is selected in a pull-down.

When designing the contents of action bars and pull-downs, it is generally desirable to group choices into major functional categories. Each category should be given a title, and these titles are used as the choices that appear in the action bar. The choices in the action bar are arranged in a horizontal list. The list can occupy multiple rows if necessary, but Common User Access recommends one only.

Common User Access contains recommendations for the contents of action bar choices and their pull-downs. The recommendations cover the wording and order of choices, and the keyboard keys that activate the choices.

Specifying Action Bar and Pull-Down Choices

The simplest way to create a list of choices is as a *menu template* in a resource script file. Using a text editor to create the resource script file, the list can be defined in ordinary text. When the compiled form of the resource script file is linked to the application, the list of choices is brought into the program using either WinCreateStdWindow, WinCreateWindow, or WinLoadMenu (WCLM). This approach is recommended, because the program code is kept simple and, because the menu definition is external, it is easier to modify, extend, or translate it. In the example code on page 5-7, the MENU statement defines the resource type as a WC_MENU class window and starts the menu definition. It can have various parameters: For example, an identifier, which can either be an integer or a symbolic that is defined in an include file as an integer, like ID_WINDOW in the example; a load option, like PRELOAD, that specifies that the resource is loaded immediately; memory options (not used in the example) that define whether the resource remains at a fixed location in memory or can be moved to optimize memory usage; and the code page in which the text is displayed.

The BEGIN and END statements are used to separate menus into functional areas. The SUBMENU statements each define a choice in the action bar, and define that each choice has a pull-down associated with it. (If you want to define an action bar choice that does not have a pull-down, you use the MENUITEM statement.)

MENUITEM and SUBMENU statements define the choices in the pull-down. MENUITEM is followed by the text string of the choice, a command value that is sent to the application in a WM_COMMAND message when the choice is selected, and can have one or more MIS_ menu item styles, followed by one or more MIA_ menu item attributes. The styles can be ORed together using the | operator, as can the attributes.

If an action needs to lead on to a more complex dialog, the text of the choice in the pull-down should be followed by an ellipsis (...) to indicate to the user that one or more dialog boxes follow. For example, here is a MENUITEM statement in the TEMPLATE.RC resource file:

```
MENUITEM "~Dialog Example...\tCtrl+D",   IDMDIALOG
```

SUBMENU statements are used when the menu item has a further pull-down associated with it. In this case the menu item has an icon, similar to a horizontal scroll bar arrow, drawn to the right of it to indicate that there is another menu to follow. Each submenu can define new submenus creating a series of cascaded menus. Following is an example of a resource definition for a three level cascading menu:

```
MENU ID_MENU1
BEGIN
  SUBMENU  "~File",                IDM_FILE
    BEGIN
      MENUITEM  "~Open... ",       IDM_OPEN
      MENUITEM  "~Close\tF3",      IDM_CLOSE
      SUBMENU   "~Save",           IDM_SAVE
      BEGIN
        MENUITEM  "~Text format", IDM_SAVETEXT
        MENUITEM  "~Sylk Format", IDM_SAVESYLK
      END
      MENUITEM  "~Quit",           IDM_QUIT
    END
    SUBMENU  "~Edit",              IDM_EDIT
  :
END
```

When TEMPLATE receives a WM_COMMAND message containing the command value IDMDIALOG, the application displays a dialog box. The actual integer value of IDMDIALOG is defined in a header file included in both the resource file and the application.

The MENU statement is similar in format to statements that define other resources in your resource script file. All the statements are listed in the *Control Program Programming Reference*. To see how the menu definition fits in with other resource definitions, see "Writing a Resource Script File" on page 8-5.

Menu Item Styles

MIS_SUBMENU
The item is an action bar choice with an associated pull-down. When the user selects the choice, a pull-down is displayed from which the user can make a further selection. Items that do not have associated pull-downs generate WM_COMMAND messages. In the resource file, you should use the SUBMENU statement, which is a functional combination of the MENUITEM statement with MIS_SUBMENU.

MIS_SEPARATOR
The display object is a horizontal dividing line. This type of item can be used only in pull-downs. This type of item cannot be enabled, checked, disabled, highlighted, or selected by the user. The functional object is NULL when this style is specified.

MIS_BITMAP
The display object is a bit map.

MIS_TEXT
The display object is a text string.

MIS_BUTTONSEPARATOR
The item is a button. Any menu may have zero, one, or two items of this type. These are the last items in a menu and are automatically displayed after a separator bar. The user cannot move the cursor to these items, but can select them with the mouse or with the appropriate key.

MIS_BREAK
The item is the last one in a row or column. The next item is displayed at the beginning of a new row or column.

MIS_BREAKSEPARATOR
This is the same as MIS_BREAK, except that it draws a separator between rows or columns of a pull-down. This style can only be used with the SUBMENU statement.

MIS_SYSCOMMAND
If this item is selected, the menu notifies the owner by posting a WM_SYSCOMMAND message rather than a WM_COMMAND message.

MIS_OWNERDRAW
Items with this style are drawn by the owner. WM_DRAWITEM and WM_MEASUREITEM notification messages are sent to the owner to draw the item or determine its size.

MIS_HELP
If the item is selected, the menu notifies the owner by posting a WM_HELP message rather than a WM_COMMAND message.

MIS_STATIC
This type of item exists for information purposes only. It cannot be selected with the mouse or keyboard.

Menu Item Attributes: Menu items have the following attributes, that applications can query or set by sending MM_QUERYITEMATTR and MM_SETITEMATTR messages.

MIA_HILITED
The state of this attribute is TRUE only if the item is selected.

MIA_CHECKED
A check mark (√) appears next to the item if the state of this attribute is TRUE.

MIA_DISABLED
This attribute is TRUE if the item is disabled and cannot be selected. The item is drawn in a disabled state.

MIA_FRAMED
If this attribute is TRUE a frame is drawn around the item.

MIA_NODISMISS
If this item is selected, the pull down containing this item is not hidden before the application window is notified of the selection.

MIA_SELECTED
This attribute is TRUE if the item is selected.

Receiving Menu Notification Messages

The action bar is usually always visible. The user can select an action bar choice using either the mouse or the keyboard. If the keyboard is used, the current choice is highlighted. The user can use the arrow keys to move to another choice.

Selecting an action bar choice causes the system to display its associated pull-down. (Pull-downs are mandatory in Common User Access.) No WM_COMMAND message is generated when the system displays a pull-down, because the application does not need to know that it has happened.

Once a pull-down has been created and displayed, you can leave the system to handle most of the processing, including all the menu drawing, inter-

actions with the user, and removing the menu. Action bars and pull-downs are implemented by a menu control window that is a child of a frame window, and is owned by it. It is worth looking at the message-based interaction between the system, and the menu control and frame window procedures, because it will help you to understand the processing that your application-provided window procedures can perform.

The user can select a choice from the pull-down, in the same way that the action bar choice is selected. When the user selects a pull-down choice, the system posts the appropriate mouse message to the application queue associated with the menu control window.

In your application, WinDispatchMsg (WMDISP) sends the mouse message to the system-provided menu control window procedure. The menu control window procedure processes the mouse message by sending an MM_SELECTITEM message to the menu control window procedure, to select the menu item. The menu control window procedure processes the MM_SELECTITEM message by sending a WM_MENUSELECT message to its owner (the frame, in this case). This indicates that a choice has been made and that the menu control window procedure intends posting a WM_COMMAND message to its owner window (the frame). (This can be a WM_SYSCOMMAND or WM_HELP message, depending on the MIS_ menu item style settings.) The message contains an identifier value, specified in the resource file, that uniquely defines the selected choice.

The frame window procedure does not process the WM_MENUSELECT message, but passes it to the default window procedure. The default window procedure returns TRUE, so that the appropriate message is posted to the frame. The frame window procedure passes the WM_COMMAND message to the FID_CLIENT window. WM_COMMAND usually contains the command value specified for the menu item, so your application can display the function for the user's choice. Chapter 6, "The HELLO2 Sample Application," and Chapter 9, "DIALOG1 Dialog Box Sample Application," and Chapter 10, "The DIALOG2 Sample Application" demonstrate how an application processes WM_COMMAND messages.

Sending and Posting Menu Control Messages

Your application can modify or query the status and other information about menu choices by sending or posting messages that have an MM_ prefix. Your application can even send MM_ messages to dynamically build and modify a menu. For example, a pull-down can be created with nothing in it, and the list of choices built up by adding one choice at a time (using WinSendMsg to send the MM_INSERTITEM message). This is a slow method of building the list and requires a large amount of code. But dynamic creation and modification can be combined with either of the other methods, which enables the application to include variable information in a menu; for example, the list of fonts available, which can differ from one system to another.

Menu Item Index: The items in a menu have a zero-based index that is used in some of the MM_ messages to specify the offset position of an item. To use the index, you specify either your own offset from zero, or the system-provided index value MIT_END for the last item in the index.

Menu Control Messages

MM_DELETEITEM
Deletes a menu item.

MM_ENDMENUMODE
Exits from menu mode and hides any active submenus.

MM_INSERTITEM
Inserts a menu item, using a MENUITEM structure to supply the index position, item style and attributes, identifier, handle of a submenu (if any) and handle of the item itself.

MM_ISITEMVALID
Returns whether an item is selectable by the user.

MM_ITEMIDFROMPOSITION
Returns the identifier of a menu item, given its index position.

MM_ITEMPOSITIONFROMID
Returns the index position, given its identifier.

MM_QUERYITEM
Copies information for a menu item into a MENUITEM structure. (Does not retrieve text, see MM_QUERYITEMTEXT.)

MM_QUERYITEMATTR
Returns the attribute settings of a menu item.

MM_QUERYITEMCOUNT
Returns the number of items in a menu.

MM_QUERYITEMTEXT
Copies the text of a menu item into a given buffer.

MM_QUERYITEMTEXTLENGTH
Returns the text length of a menu item.

MM_QUERYSELITEMID
Returns the identifier of a menu item in a menu or submenu, if one is selected.

MM_SELECTITEM
Selects or deselects a menu item.

MM_SETITEM
Copies information from MENUITEM structure into a menu item. (Does not copy text, see MM_SETITEMTEXT.)

MM_SETITEMATTR
Sets menu item attributes.

MM_SETITEMHANDLE
Sets a menu item handle.

MM_SETITEMTEXT
Sets the text of a menu item.

MM_STARTMENUMODE
Begins menu selection processing.

The MM_ messages are described in full in the *Programming Reference: Volume 2*

Disabling and Enabling Menu Update

Normally, all MM_ messages that change the menu cause an immediate redraw of the menu. If multiple changes need to be made at one time, then the changes can be made with only one screen update, using the following technique:

1. Invoke the WinEnableWindowUpdate (WIEWU) function with its NewVisibility parameter set to FALSE for the menu control window, to prevent that window being redrawn on each change.
2. Send the collection of messages necessary to change the menu control.
3. Send a WM_FORMATFRAME message to the frame window containing the menu control, indicating that the menu control requires redrawing.
4. Invoke the WinEnableWindowUpdate function with its its NewVisibility parameter set to TRUE for the menu control window, to ensure that that window is enabled for redrawing.

A pull-down is *dismissed* (removed from the screen) in one of two ways:

- The default behavior (in conformance with Common User Access), is for a pull-down to be dismissed automatically when the user selects a choice or presses Escape.
- For MIA_NODISMISS choices, the pull-down is dismissed at some later point, determined by the program. In this case, the application must send an MM_DISMISSMENU control message to the pull-down.

Example Code: Sending Menu Control Messages

The IMAGE sample application sends an MM_SETITEMATTR message to enable or disable a menu item, according to the contents of a variable option:

```
WinSendMsg( WinWindowFromID( hwndFrame, FID_MENU)
          , MM_SETITEMATTR
          , MPFROM2SHORT( usitem, TRUE)
          , MPFROM2SHORT(MIA_DISABLED, fOption ? ~MIA_DISABLED :
                                                 MIA_DISABLED));
```

Creating a Menu Template in a Program

A menu can also be created as a menu template data structure in program memory, using the WinCreateMenu call. The menu template data structure is more complex than a resource file description, but does the same job. WinCreateMenu allows menus to be built and modified dynamically, and gives better performance than the method of sending MM_ messages. However, you are unlikely to need this function.

Note: This call cannot be used in COBOL/2 and FORTRAN/2.

Menu Mnemonics

A mnemonic is a keyboard key that causes direct selection of an action bar or pull-down choice, without the user having to move the cursor to the choice. A mnemonic operates only when the input focus is in the window for which it is defined (the action bar or pull-down), and applies only to that window.

The sequence for the keyboard user is:

1. Activate the action bar (F10 from the client area).
2. Enter the mnemonic for an action bar choice. This selects the choice and causes the associated pull-down to appear. It also switches the input focus to the pull-down.
3. Enter the mnemonic for a pull-down choice. This selects the choice and causes its associated action to take place.

A mnemonic key must be one of the letters used in the text of the choice. The program defines the mnemonic letter by putting a ~ character in front of it in the text following the MENUITEM or SUBMENU statement in the resource file. For example, the letter E is made the mnemonic in the **Example** choice in TEMPLATE, as follows:

```
SUBMENU  "~Example", IDMEXAMPLE
```

The ~ character is not displayed in the action bar or pull-down that the user sees. Instead, the mnemonic letter is underscored in the text of the choice. You cannot display a ~ character in a menu.

Accelerators

Normally, when the user presses a keyboard key, OS/2 sends a WM_CHAR message to the active window. But, when the user presses a key defined as an accelerator key in the resource file, OS/2 converts the WM_CHAR message into a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message, depending on what is specified in the resource file. The original WM_CHAR message no longer exists, and the new message is directed to the active window. To load the accelerator table from the resource file, specify the FCF_ACCELTABLE frame creation flag (or FCF_STANDARD).

When the active window is of the WC_FRAME class (usually the case) the system-provided WC_FRAME window procedure, processes each message as follows:

WM_COMMAND
The WC_FRAME window procedure sends this to the FID_CLIENT window if one exists. If no FID_CLIENT window exists, the message is sent to the default window procedure, which ignores it.

WM_SYSCOMMAND
The WC_FRAME window procedure takes the appropriate action if it recognizes one of the SC_ values. Otherwise, it discards the message.

WM_HELP
The WC_FRAME window procedure handles this independently of whether it was received as the result of an accelerator.

An accelerator key works in a similar way to a function key. However, a function key is usually one of the Fn keys on the keyboard, for example F1. By contrast, an accelerator key can be any key on the keyboard, and is usually displayed by a choice in a pull-down. Accelerator keys work whenever the related window is active; the pull-down that displays the keys does does not need to be active. Accelerators are a fast path method to be used by experienced users of a program.

You usually use Alt, or Ctrl, with alphanumeric keys, or function keys to distinguish them from the commonly used alphanumeric keys. The accelerator key related to a particular choice should be displayed in the pull-down, to the right of the choice. This is done by putting \t after the text in the resource script file, This is followed by the name

of the accelerator key, for example, F3, as in the following example:

```
MENUITEM    "E~xit \tF3",  IDMEXIT
```

The text separated in this way is right-justified in the pull-down, with at least a single space between it and the main text. Here is the accelerator table definition used in TEMPLATE.RC:

```
ACCELTABLE      ID_TEMPLATE
BEGIN
    VK_F3,      IDMEXIT,    VIRTUALKEY
    "^D",       IDMDIALOG
END
```

An accelerator can apply either to one window only (for example, a main window) or to the whole application. There is a two-level accelerator table structure. One table applies at window level, for the active window. The other table applies at queue level, for all windows belonging to one input queue, and is set by the WinSetAccelTable (WMSACC) call. This table contains the system-provided accelerators for the system menu. Each table is searched in sequence when a key is pressed, window table first. This allows a key to have a general meaning at queue level and a different meaning in some windows. If a match is found in the window table, it is translated into a WM_COMMAND message at once and the queue table is not searched.

When the user presses a key that is not defined as a mnemonic or accelerator, the system sounds a warning beep, unless the user has used the control panel of the user shell to turn off the warning beep.

Action Bar and Pull-Down Conventions

Common User Access recommends standard contents for File, Edit, and Help pull-downs. The recommendations cover the wording and order of choices, and their mnemonics and accelerators. Common User Access provides guidance for View and Options pull-downs.

File Action Bar Item and Pull-Down: When an application needs to load and save files, it provides a **File** choice as the first choice in its action bar. The subsequent pull-down contains the following choices:

Open
: Displays a dialog box containing a list of directories and files, from which the user is prompted to select. Your application should load a copy of the selected file.

New
: Clears the current file and resets an initial state.

Save
: Saves a copy of the current file on disk.

Save as
: Displays a dialog box containing listbox and an entry field. The list box allows the user to choose the drive and directory in which the file is to be saved and the entry field allows the user to type in a file name.

Print
: Prints the current file.

Exit(F3)
: Closes the window and ends the application.

Edit Action Bar Choice and Pull-Down: Each application using the clipboard provides an **Edit** choice in its action bar. The subsequent pull-down contains the following choices:

Undo (Alt+Backspace)
: Return to the state before the previous action, if possible.

Cut (Shift+Del)
: Removes the selected data and places it in the clipboard.

Copy (Ctrl+Ins)
: Places a copy of the selected data in the clipboard.

Paste (Shift+Ins)
: Inserts the data in the clipboard in the selected field.

Clear (Del)
: Removes the currently selected text, but does not place it in the clipboard.

You may also add other choices; for example, to select data before editing, or to undo previous actions.

Help Action Bar Item: Common User Access requires an application to supply online help information. You should provide help that is organized to be available and relevant to the functional levels in your application. Refer to the Common User Access publications for full details on help requirements.

To provide help, your application can either:

- Process the WM_HELP message, as in the DIALOG2 example.
- Provide a *help hook procedure*. See "Hooks" on page 13-19 for details.
- Use the OS/2-provided Help Manager. The Help Manager is a programming tool that you can use to specify and associate help panels with your application panels and fields. Your application subsequently calls the Help Manager, which displays the help panels for your application in response to user requests. The Help Manager saves time and increases productivity because you no longer have to code the processing of help requests as part of the application logic. In addition, Common User Access rules are built into the Help Manager, making it easy to create and maintain consistency among all of your help panels. The Help Manager is described in "Part 3. Presentation Manager Applications – Using OS/2 Information Presentation Facility."

For the Help Manager, you must define the Help pull-down such that selecting 'Help index...', 'Extended Help...' and 'Keys Help...' generate WM_SYSCOMMANDS with command values of SC_HELPINDEX, SC_HELPEXTENDED and SC_HELPKEYS respectively.

System Menu Icon

The system menu icon is actually a single item in a WC_MENU class window. If the user selects the system icon, which is located in the upper-left corner, a pull-down appears, offering the user various OS/2-related choices. You obtain the icon and menu using the FCF_SYSMENU flag (or FCF_STANDARD). From the system menu, the user can exit from the application. The system menu provides a similar set of functions for window sizing for the keyboard user. Here, the window size changes in response to keyboard arrow keys, after the sizing operation is started.

The system menu is a system-wide menu, so all applications in the system see the same basic menu. If any of the system menu choices are not applicable to your application, you can disable them by sending a MM_SETITEMATTR message with the MIA_DISABLED menu item attribute. You can also use the MM_INSERTITEM message to add your own items to the system menu.

Sizing Borders

The *sizing borders* let the user change the size of the frame window to an almost arbitrary size. To obtain them, use the FCF_SIZEBORDER (for a wide border) or FCF_BORDER (for a thin border) frame creation flags. The border is in eight segments – four sides and four corners. To change the size of the window, the user uses the press and hold method with the mouse. Picking a side changes the size of the window by moving only that side when the mouse moves. Picking a corner changes the size by moving the two sides attached to the corner.

While the mouse button is pressed down, the size of the window is represented on the screen by a *shadow box*. The shadow box follows the mouse movements and shows the user what shape the window takes if the mouse button is released. The frame window is changed in size when the user releases the mouse button. The application is informed that the frame window size has changed, by a WM_SIZE message that is sent to the FID_CLIENT window.

Repainting as a Result of Sizing

Your application is responsible for adjusting the data in its own windows as a result of sizing actions. OS/2 does not automatically do it for you. If the application does nothing, the data is clipped off at the edge of the window. This may leave some important parts of the data invisible.

The application should respond to a change of size, which is indicated by the WM_SIZE message, by either rescaling the data to fit into the new window size, or adjusting the layout of the data to ensure that important parts are still visible.

Usually, you should not put code in your application to reformat the data in windows that are partially or completely overlaid by other windows. Reformatting data under these circumstances usually only confuses the user.

Scroll Bars

If the data displayed in the window is greater than the physical display space available in the window, the user needs to be able to scroll the data through the window to access whichever part of the data is required. For example, when a long file is displayed in a window, and the full contents of the file are too large to fit on the screen, the user has to choose which section of the file to view. To provide the user with the means to scroll data in a standard window, you have the option of providing scroll bars.

A scroll bar is a rectangular window, that can be either horizontally or vertically oriented. Within the scroll bar are the following elements:

1. A movable slider within a central rectangular area that constitutes most of the scroll bar. The position of the slider within the slider area provides a visual indication to the user of the position of the displayed data in relation to the total amount of data available. For example, the position of the slider can indicate how far he or she has scrolled through a long piece of text. The size of the slider compared to the slider area provides a visual indication of the size of the displayed data in relation to the total amount of data available.
2. Slider areas on either side of the slider itself.
3. Single arrows at each end of the scroll bar. These arrows let the user move the slider position, usually by a small amount set by the application.

The standard window can provide the user with vertical or horizontal scroll bars, or both. The scroll bars usually appear to the right and bottom of the client window. Their appearance is controlled by the FCF_HORZSCROLL and FCF_VERTSCROLL flags when the window is created.

You can also create scroll bar control windows as children of any window, as described in "Scrolling Panel Contents – Scroll Bars" on page 7-18. OS/2 also uses them in another control window class, as described in "Scrollable Selection Fields – List Boxes" on page 7-10.

A horizontal scroll bar is a WC_SCROLLBAR class window, with an SBS_HORZ window style. When it is a component of the standard window, it has the identifier FID_HORZSCROLL. A vertical scroll bar in a standard window is a WC_SCROLLBAR class window, with an SBS_VERT window style. When it is a component of the standard window, it has the identifier FID_VERTSCROLL.

The window style SBS_THUMBSIZE gives you a variable-size slider.

Receiving Scroll Bar Notification Messages

The user can request scrolling using the mouse or keyboard. When the user interacts with a scroll bar in a standard window, a WM_HSCROLL message (horizontal scrolling) or WM_VSCROLL message (vertical scrolling) is sent to the frame window, which passes the message on to the FID_CLIENT window. The WM_HSCROLL and WM_VSCROLL messages contain command values, with an SB_ prefix, that indicate the part of the scroll bar with which the user is interacting.

The WM_HSCROLL command values, also shown in Figure 5-3 on page 5-17, are as follows:

SB_LINELEFT
The user has clicked the left arrow of the scroll bar, or pressed the VK_LEFT key. (See the *Control Program Programming Reference* for a list of the VK_ keys.)

SB_LINERIGHT
The user has clicked the right arrow of the scroll bar, or pressed the VK_RIGHT key.

SB_PAGELEFT
The user has clicked the area to the left of the slider, or pressed the VK_PAGELEFT key.

SB_PAGERIGHT
The user has clicked the area to the right of the slider, or pressed the VK_PAGERIGHT key.

SB_SLIDERPOSITION
Indicates the final position of the slider, when the user finishes scrolling.

SB_SLIDERTRACK
The user is moving the slider with the pointer.

SB_ENDSCROLL
Sent when the user finishes scrolling, and has not been using the slider.

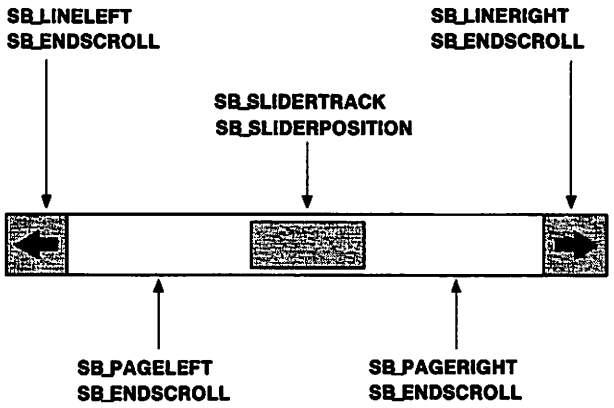


Figure 5-3. Horizontal Scroll Bar

The WM_VSCROLL command values, also shown in Figure 5-4, are as follows:

SB_LINEUP
The user has clicked the up arrow of the scroll bar, or pressed the VK_UP key.

SB_LINEDOWN
The user has pressed the down arrow of the scroll bar, or pressed the VK_DOWN key.

SB_PAGEUP
The user has clicked the area above the slider, or pressed the VK_PAGEUP key.

SB_PAGEDOWN
The user has pressed the area below the slider, or pressed the VK_PAGEDOWN key.

SB_SLIDERPOSITION
Indicates the final position of the slider, when the user finishes scrolling.

SB_SLIDERTRACK
The user is moving the slider with the pointer.

SB_ENDSCROLL
Sent when the user finishes scrolling, and has not been using the slider.

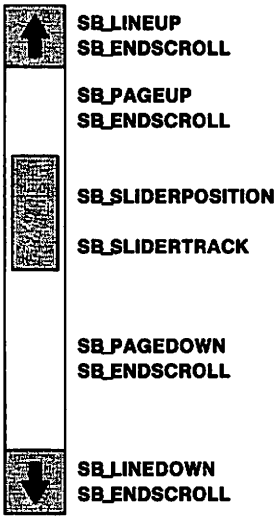


Figure 5-4. Vertical Scroll Bar

Your application can interpret the command values as follows:

1. When the slider is released, your application can determine the amount of data to display from the new position of the slider.
2. Arrows or arrow keys should move the data by the *smallest increment*. The smallest increment is defined by the application: for text, it is usually one line vertically or one character horizontally.
3. Selecting the area on either side of the slider should cause a paging action. The size of the page scroll depends on the application: a text document should scroll through the window by the amount that can be seen in the window, less one line. The subtraction of one line from the amount ensures that, for example, when the user is paging down through a document, the bottom line in the window before scrolling becomes the top line after scrolling.

You might want keys other than the VK_ keys above to cause scrolling. The FID_CLIENT window receives key input in WM_CHAR messages, which your application can process. For example, to process the WM_CHAR message, your application could send itself a WM_HSCROLL or WM_VSCROLL message. The message would be dealt with in the same way as mouse-initiated scrolling. Alternatively, both the processing of the WM_CHAR message and WM_VSCROLL or WM_HSCROLL messages could call a scrolling subroutine in your application.

Example Code: Receiving Scroll Bar Notification Messages

The following example shows a simplified version of the processing of WM_VSCROLL from the AVIOSAMP sample program. Each notification code is processed by calling the Scroll scrolling subroutine.

```
WM_VSCROLL:
  :
switch( HIUSHORT( mp2))
   {
    case SB_LINEDOWN:
        Scroll( 0, WNDROW );
        break;

    case SB_LINEUP:
        Scroll( 0, -WNDROW );
        break;

    case SB_PAGEDOWN:
        Scroll( 0, WndDepth-2 );
        break;

    case SB_PAGEUP:
        Scroll( 0, -( WndDepth-2 ));
        break;

    case SB_SLIDERPOSITION:
        Scroll( 0,  LOUSHORT( mp2 ) - orig.y );
        break;

    default:
        break;
    }
        }
```

Sending Scroll Bar Control Messages

When a window needs to control the status of a scroll bar, it can send it any one of several control messages that have an SBM_ prefix, where SBM stands for scroll bar message. The control messages for scroll bars are as follows:

SBM_QUERYPOS
Returns the slider position.

SBM_QUERYHILITE
Returns the highlighting status.

SBM_QUERYRANGE
Returns the scroll bar range.

SBM_SETHILITE
Sets the highlighting status. When the user has scrolled as far as is possible in one direction, your application should gray the scroll bar arrows that point in that direction.

SBM_SETPOS
Sets the slider position When the user completes a scrolling operation, your application should reset the position of the slider.

SBM_SETSCROLLBAR
Sets the scroll bar range and slider position.

SBM_SETTHUMBSIZE
For a scroll bar with the window style SBS_THUMBSIZE, supplies the number of items (for example, lines in a document) that are visible in a window, and the total number of available items. From these, the system calculates the percentage of the slider area that the slider should occupy, and draws the slider accordingly. The application usually sends this message either when initializing the scroll bar, or when its client window is changing in size. When the user sizes a window containing a scroll bar, or alters the total amount of available items, your application should reset the size of the slider.

Example Code: Sending Scroll Bar Control Messages

The AVIOSAMP sample program sends an SBM_SETPOS message like this to set the position of the scroll bar slider:

```
WinSendMsg( WinWindowFromID( hwndFrame,
            FID_VERTSCROLL ),
            SBM_SETPOS,
            MPFROMLONG( Origin.y+1 ),
            MPFROMSHORT( 0 )
            );
```

Repainting as a Result of Scrolling

Your application is responsible for adjusting the data in its own windows as a result of scrolling actions. OS/2 does not automatically do it for you.

One method of scrolling is to move the origin of the presentation space, and to repaint the window. This is done in the AVIOSAMP sample program.

Another method is to use the WinScrollWindow (WISCRL) call. WinScrollWindow scrolls *only the data that is in the window*. It does *not* automatically scroll data that is *outside* the window *into* the window, because such data has already been clipped at the window boundary. Your application is responsible for repainting the data to fill in the gap left by scrolling. In other words, if the user scrolls some graphics text up, and your application does nothing, then a gap is left at the bottom of the window. Your application can use the update region to discover which parts of the window it should fill with the missing data. This is done in the GRAPHIC1 and GRAPHIC2 sample programs.

Client Window

Your application displays its data in the FID_CLIENT window in the client area. What is displayed in the FID_CLIENT window obviously depends on the function that your application is providing to the user. As already mentioned, you can draw your own graphics or alphanumerics in the FID_CLIENT window. You can subdivide the client area by creating windows that are children of the FID_CLIENT window. These can, in turn, have their own child windows, and so on. Child windows can be windows in which you draw your own graphics or alphanumerics, or they can be windows of system-provided window classes, as described in Chapter 7, "Dialog Boxes and Controls."

The FID_CLIENT window can receive messages from a variety of sources:

- Messages that are the result of user interaction with the frame control windows of the standard window can be passed on to the FID_CLIENT window by the frame window. Examples are the scrolling messages mentioned above.
- The FID_CLIENT window can receive mouse messages when the mouse is over it. The FID_CLIENT window can receive keyboard messages when the FID_CLIENT window has the input focus.
- The FID_CLIENT window can receive messages that are implicitly or explicitly sent by calls in your program.
- The FID_CLIENT window can receive messages from the system as a result of user interaction with other windows. For example, when a window loses the input focus as a result of another window gaining the input focus, the window losing the focus is sent the WM_FOCUSCHANGE message.

You must decide whether your window procedure needs to process a message explicitly, or whether it can pass it to the default window procedure. Which messages your program processes is determined by the function of your program, and the degree of control you want your application to exercise in responding to the messages. In many cases, the processing provided by the default window procedure is adequate.

The processing of messages by the frame window and by the default window procedure are documented in the *Programming Reference: Volume 2.*

There may also be occasions when another window is sent a message that your FID_CLIENT window needs to intercept. In this case, your application must subclass the window to intercept the message, as described in "Subclassing a Window" on page 13-9.

Adding Controls to the Standard Window

The standard window provides much function that can be used in many application programs. Also, like other parts of the OS/2 API, the function can be added to without losing any of the standard function already available.

In particular, it is possible to add additional windows or controls to the standard set. Examples of this include providing a command line at the bottom of the client area, splitting the client area into two or more windows to display different kinds of data in each window, and placing a page number field alongside the horizontal scroll bar at the bottom of the window.

To perform reformatting of the standard window as a result of sizing, the window procedure of the client window should subclass the frame window to intercept the WM_FORMATFRAME message that the frame sends itself whenever it receives a WM_SIZE message. WM_FORMATFRAME contains a pointer to an array that contains the positions and sizes of all the standard controls of the frame window. The application can add additional window positions and sizes to the array and modify the existing ones, to produce a layout that includes some nonstandard frame controls or modifies the standard ones. The array is passed to WinSetMultWindowPos (WISMPS), which sets the size and position of all the windows referenced by the array.

In the example suggested above of displaying a page number alongside the horizontal scroll bar, the application would do the following:

1. Add the window size and position for the page number control at the lower left of the frame window
2. Modify the scroll bar size by making it smaller in the horizontal direction
3. Move the origin of the scroll bar to the right by the width of the page number control.

Chapter 6. The HELLO2 Sample Application

The HELLO2 application is based on HELLO1, but provides a more sophisticated user interface that displays an SAA Common-User-Access-conforming standard window consisting of a frame and a panel body area containing the FID_CLIENT window. It also uses a resource file to define the contents of the action bar, the pull-down, and the accelerators. The frame in the example contains several standard visual elements that are specified by your application but drawn and controlled for you by OS/2. The frame encloses the client area.

The frame in the example contains the following elements:

System menu icon
Located in the upper-left corner of the window. When selected, displays a system menu.

Title bar
Contains the name of the window.

Maximize/minimize/restore buttons
Located in the upper-right corner of the window. The user changes the size of the window with the buttons.

An action bar
Contains a horizontal menu with the single choice **Options**. Selecting the choice causes the pull-down to be superimposed on the first window, below the action bar. The pull-down contains a vertical menu, allowing a further level of selection. The pull-down contains four choices, representing three functions of the application and an **Exit** option that provides a visible means for the user to exit or resume the application. Each function displays one of three text strings in the window instead of **Hello**. When the user selects one of the choices, a message is posted to the application, and one of the three alternative text strings is displayed. The application provides keyboard key equivalents (called *accelerator keys*) for the three functions in the pull-down, as a fast path alternative to the menu. When the user presses a key that is defined as an accelerator key, a WM_COMMAND message is posted to the application. The message has the same content as the message sent when a choice is selected.

OS/2 implements the frame of a standard window by creating a *frame window*, and implements the client area by creating a *client window* as a child of the frame window. No part of the frame window is obscured, because OS/2 draws the client window inside the frame. In the client window, your application can display its output or create further windows.

Standard windows are described in Chapter 5, "The Standard Window."

The visual appearance of the application is shown in Figure 6-1. The **Options** pull-down is shown.

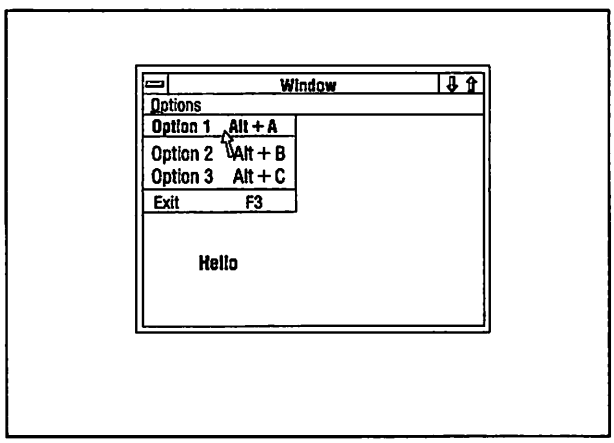


Figure 6-1. Appearance of Screen Displayed by HELLO2

The Resource Script File

The definitions for the menus and for the accelerator keys are held in a resource file, separate from the C/2 source file for the main procedure and window procedure. The resource file is initially defined as a *resource script file* – a simple ASCII file similar to a source code file. A compiler called the *Resource Compiler* compiles the resource script file into an object-form resource file, and binds it to your application's .EXE file.

Example Code: HELLO2 Resource Script File

The resource script file for HELLO2 is as follows:

```
#include <os2.h>
#include "hello2.h"

ICON    ID_WINDOW hello2.ico

MENU    ID_WINDOW PRELOAD
BEGIN
  SUBMENU "~Options", ID_OPTIONS
  BEGIN
    MENUITEM "Option ~1\tAlt+a", ID_OPTION1, MIS_TEXT
    MENUITEM "Option ~2\tAlt+b", ID_OPTION2, MIS_TEXT
    MENUITEM "Option ~3\tAlt+c", ID_OPTION3, MIS_TEXT
    MENUITEM SEPARATOR
    MENUITEM "E~xit \tF3",       ID_EXITPROG, MIS_TEXT
  END
END

ACCELTABLE ID_WINDOW PRELOAD
BEGIN
  "a",   ID_OPTION1, CHAR,        ALT
  "b",   ID_OPTION2, CHAR,        ALT
  "c",   ID_OPTION3, CHAR,        ALT
  "A",   ID_OPTION1, CHAR,        ALT
  "B",   ID_OPTION2, CHAR,        ALT
  "C",   ID_OPTION3, CHAR,        ALT
  VK_F3, ID_EXITPROG, VIRTUALKEY
END

STRINGTABLE PRELOAD
BEGIN
 IDS_HELLO, "Hello"
 IDS_1,     "--1--    "
 IDS_2,     "--2--    "
 IDS_3,     "--3--    "
END
```

The menus, accelerator keys, and character strings are defined in this single file. In general, any number of every type of resource can be held in a single resource file. For a fuller explanation of the contents of the resource script file, see "Writing a Resource Script File" on page 8-5. The syntax of the statements is given in the *Control Program Programming Reference*.

You can see that many of the choices displayed on the screen by the program are defined in the resource script file. Most of the resources and the items they contain have symbolic identifiers, all starting with ID_, and used by the program to refer to the resources.

The resource script file includes the OS2.H file. It also includes the file hello2.h, where all the symbolic names and their numeric values are kept. hello2.h looks like this:

```
#define ID_WINDOW   256

#define ID_OPTIONS  257
#define ID_OPTION1  258
#define ID_OPTION2  259
#define ID_OPTION3  260

#define ID_EXITPROG 261

#define IDS_HELLO   262
#define IDS_1       263
#define IDS_2       264
#define IDS_3       265
```

In the resource script file, the ICON statement defines the file HELLO2.ICO. The file contains an icon that the window uses to represent itself when reduced to its smallest size. The icon resource is given the identifier ID_WINDOW.

The MENU statement defines the start of the action bar menu, and also gives it the identifier ID_WINDOW. The identifier is the same as that of the icon, because the program uses ID_WINDOW to refer to all the resources relating to one window.

The SUBMENU statements specify the action bar choices **Commands** and **Exit**, which each have pull-downs. The MENUITEM statements define the menus in the pull-downs.

Each MENUITEM statement holds the definition of the text for the choice, the symbolic identifier of the choice. Like the SUBMENU statements, it also specifies that the choice is a text string. Other types of choice, such as bit maps, are possible in sophisticated applications.

The SUBMENU and MENUITEM statements have symbolic identifiers. When the user selects a choice relating to a MENUITEM statement, OS/2 posts a WM_COMMAND message to the application. The message contains the numeric value of the symbolic identifier of the selected choice. The values of the symbolic identifiers of the choices in the SUBMENU statements are not received by the application. When the user selects these choices, a WM_COMMAND message is *not* generated, but OS/2 displays the pull-down.

In the SUBMENU and MENUITEM statements, the tilde character (~) denotes the character that is the mnemonic for each choice.

For the **Options** pull-down, each of the choices has a \t in the middle of the text string, which is not actually displayed in the menu. This is a tab mark, and is used to tabshift the following text. It is justified to the right of the main text of the choice. The text that follows the \t in each case shows the accelerator keys that the user presses to select the choices.

The accelerator key definitions are introduced in the resource script file by the ACCELTABLE statement, which again has an identifier of ID_WINDOW. It can be loaded by the same function call as the menu. Each line in the definition consists of an ASCII value (in quotes), an identifier value that is passed to the application in the WM_COMMAND message when the key is pressed, and finally the shift states that apply to the key; Alt, in the example.

Finally, you can define a table of text strings, using the STRINGTABLE statement. You give each string its own identifier, and load it into the application separately.

Example Code: HELLO2 Sample PM Application

The source code for HELLO2.C is:

```
#define INCL_GPIPRIMITIVES              /* Selectively include          */
#define INCL_WINFRAMEMGR                /* relevant parts of            */
#define INCL_WINSYS                     /* the PM header file           */

#include <os2.h>                        /* PM header file               */
#include <string.h>                     /* C/2 string functions         */
#include "hello2.h"                     /* Resource symbolic identifiers*/

#define STRINGLENGTH 20                 /* Length of string             */

/****************************************************************************/
/* Function prototypes                                                      */
/****************************************************************************/
MRESULT EXPENTRY MyWindowProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 );
void cdecl main( void );

                                        /* Define parameters by type    */
HAB  hab;                               /* PM anchor block handle       */
CHAR szHello[STRINGLENGTH];             /* String parameters, set in    */
CHAR sz1[STRINGLENGTH];                 /* the processing of WM_CREATE, */
CHAR sz2[STRINGLENGTH];                 /* and used in the processing   */
CHAR sz3[STRINGLENGTH];                 /* of WM_COMMAND, in window     */
CHAR szString[STRINGLENGTH];            /* procedure.                   */

/*********************  Start of main procedure  ***********************/
void cdecl main(  )
{
  HMQ  hmq;                             /* Message queue handle         */
  HWND hwndClient;                      /* Client area window handle    */
  HWND hwndFrame;                       /* Frame window handle          */
  QMSG qmsg;                            /* Message from message queue   */
  ULONG flCreate;                       /* Window creation control flags*/

  hab = WinInitialize( NULL );          /* Initialize PM                */
  hmq = WinCreateMsgQueue( hab, 0 );    /* Create a message queue       */

  WinRegisterClass(                     /* Register window class        */
     hab,                               /* Anchor block handle          */
     "MyWindow",                        /* Window class name            */
     MyWindowProc,                      /* Address of window procedure  */
     CS_SIZEREDRAW,                     /* Class Style                  */
     0                                  /* No extra window words        */
     );

  flCreate = FCF_STANDARD &             /* Set Frame Control Flags to   */
             ~FCF_SHELLPOSITION;         /* standard except for shell    */
                                        /* positioning.                 */

   hwndFrame = WinCreateStdWindow(
               HWND_DESKTOP,            /* Desktop window is parent     */
               0L,                      /* No  frame styles             */
               &flCreate,               /* Frame control flag           */
```

```
                "MyWindow",             /* Client window class name     */
                "",                     /* No window text               */
                0L,                     /* No special class style       */
                NULL,                   /* Resource is in .EXE file     */
                ID_WINDOW,              /* Frame window identifier      */
                &hwndClient             /* Client window handle         */
                );

  WinSetWindowPos( hwndFrame,           /* Shows and activates frame    */
                   HWND_TOP,            /* window at position 100, 100, */
                   100, 100, 200, 200,  /* and size 200, 200.           */
                   SWP_SIZE | SWP_MOVE | SWP_ACTIVATE | SWP_SHOW
                 );


/*********************************************************************/
/* Get and dispatch messages from the application message queue      */
/* until WinGetMsg returns FALSE, indicating a WM_QUIT message.     */
/*********************************************************************/
  while( WinGetMsg( hab, &qmsg, NULL, 0, 0 ) )
    WinDispatchMsg( hab, &qmsg );

  WinDestroyWindow( hwndFrame );        /* Tidy up...                   */
  WinDestroyMsgQueue( hmq );            /* and                          */
  WinTerminate( hab );                  /* terminate the application    */
}
/*********************  End of main procedure  ************************/

/*******************  Start of window procedure  *********************/
MRESULT EXPENTRY MyWindowProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
  USHORT command;                       /* WM_COMMAND command value     */
  HPS    hps;                           /* Presentation Space handle    */
  RECTL  rc;                            /* Rectangle coordinates        */
  POINTL pt;                            /* String screen coordinates    */

  switch( msg )
  {
    case WM_CREATE:
      /*****************************************************************/
      /* Window initialization is performed here in WM_CREATE processing*/
      /* WinLoadString loads strings from resource file                */
      /*****************************************************************/
      WinLoadString( hab, NULL, IDS_HELLO, STRINGLENGTH, szHello );
      WinLoadString( hab, NULL, IDS_1,     STRINGLENGTH, sz1     );
      WinLoadString( hab, NULL, IDS_2,     STRINGLENGTH, sz2     );
      WinLoadString( hab, NULL, IDS_3,     STRINGLENGTH, sz3     );
      strcpy( szString, szHello );      /* Copy text Hello into szString*/
      break;

    case WM_COMMAND:
      /*****************************************************************/
      /* For the Commands pull down, when the user chooses option 1, 2, */
      /* or 3, the text string is set to 1, 2, or 3, and               */
      /* WinInvalidateRegion sends a WM_PAINT message.                 */
      /* For the Exit pull down, the application posts itself a WM_CLOSE*/
      /* message if Exit is chosen, or resumes if Resume is chosen.    */
      /*****************************************************************/
      command = SHORT1FROMMP(mp1);      /* Extract the command value    */
      switch (command)
```

```
    {
      case ID_OPTION1:
        strcpy( szString, sz1 );
        WinInvalidateRegion( hwnd, NULL, FALSE );
        break;
      case ID_OPTION2:
        strcpy( szString, sz2 );
        WinInvalidateRegion( hwnd, NULL, FALSE );
        break;
      case ID_OPTION3:
        strcpy( szString, sz3 );
        WinInvalidateRegion( hwnd, NULL, FALSE );
        break;
      case ID_EXITPROG:
        WinPostMsg( hwnd, WM_CLOSE, 0L, 0L );
        break;
      default:
        return WinDefWindowProc( hwnd, msg, mp1, mp2 );
    }
    break;
  case WM_ERASEBACKGROUND:
    /*********************************************************************/
    /* Return TRUE to request PM to paint the window background          */
    /* in SYSCLR_WINDOW.                                                 */
    /*********************************************************************/
    return (MRESULT)( TRUE );
  case WM_PAINT:
    /*********************************************************************/
    /* Window contents are drawn here in WM_PAINT processing.            */
    /*********************************************************************/
                                    /* Create a presentation space  */
    hps = WinBeginPaint( hwnd, NULL, &rc );
    pt.x = 50; pt.y = 50;           /* Set the text coordinates,    */
    GpiSetColor( hps, CLR_NEUTRAL );        /* color of the text,   */
    GpiSetBackColor( hps, CLR_BACKGROUND ); /* its background and   */
    GpiSetBackMix( hps, BM_OVERPAINT );     /* how it mixes,        */
                                            /* and draw the string...*/
    GpiCharStringAt( hps, &pt, (LONG)strlen( szString ), szString );
    WinEndPaint( hps );                     /* Drawing is complete  */
    break;
  case WM_CLOSE:
    /*********************************************************************/
    /* This is the place to put your termination routines                */
    /*********************************************************************/
    WinPostMsg( hwnd, WM_QUIT, 0L, 0L );  /* Cause termination           */
    break;
  default:
    /*********************************************************************/
    /* Everything else comes here.  This call MUST exist                 */
    /* in your window procedure.                                         */
    /*********************************************************************/

    return WinDefWindowProc( hwnd, msg, mp1, mp2 );
  }
  return FALSE;
}
/*********************  End of window procedure  *********************/
```

Additional Include Files

There are two additional include files in HELLO2, string.h and hello2.h. string.h is a C function include file containing string-handling functions, one of which, strcpy, is used several times in the program. hello2.h contains the symbolic identifiers of the resources.

Declarations of Character Variables

The CHAR statements are C/2 statements declaring character variables used for text strings later in the program.

Changes to the Main Procedure

WinRegisterClass (WMRCL) registers a window class of MyWindow, as in HELLO1. Here, it specifies a class style of CS_SIZEREDRAW, which tells OS/2 to redraw the contents of the window if the user changes its size.

For simplicity, HELLO2 uses WinCreateStdWindow to create the application's window inside a standard frame window, which holds sizing borders, a title bar, and menus. The frame window is a window of a system-provided window class. Therefore, the application does not have to supply any code to obtain the function for this type of window. The client window is a child window of the frame.

Two window handles are returned by this function, a handle for the frame window and a handle for the client window. Each of these can be used in subsequent operations.

The frame is created invisible. Therefore, the client window is invisible at first, as child windows inherit the visible state of their parent windows. This also applies to all the menu windows and title bar.

The frame could have been created visible. In this case, the WinCreateStdWindow function would have created the frame with a size and position determined by the User Shell, with reference to any other windows on the screen. To allow the application to control the window's position and size itself, the window is created invisible. A size of zero and a position of zero in both dimensions are allocated to the window. The application is responsible for setting the size and position of the frame window and making it visible. This is the function of WinSetWindowPos (WISPOS), described below.

The application specifies the FCF_STANDARD frame creation flag, which is the same as combining several of the available FCF_ flags. The FCF_SHELLPOSITION flag is excluded, because the size and position of the window are set by the application, not the User Shell.

When the WinCreateStdWindow (WICRTS) function needs to create a menu and load accelerator key definitions, it looks for the definitions of these choices in a resource file associated with the application. The resource file is defined by the third from last parameter to the function. In the example this is set to NULL, which specifies that the resource file is in the application's .EXE file, as described in "Creating a Resource File and Binding It to Your .EXE File" on page 8-6. Alternatively, the parameter can contain the name of a dynamically linked module containing the resource file, as described in "Putting a Resource File in a Dynamic Link Library Module" on page 8-10.

A resource file can contain many resources of the same type, for example, many menus. The resources are distinguished by a resource identifier that is unique for each resource of a particular type. The second to last parameter to WinCreateStdWindow gives the resource identifier ID_WINDOW. ID_WINDOW matches the resource identifier used for the icon, menu definition, and the accelerator key definitions in the resource script file.

Like HELLO1, when HELLO2 creates its window, it does *not* use the WS_VISIBLE style. The window is, therefore, created invisible, and must subsequently be made visible by calling WinSetWindowPos. WinSetWindowPos (WISPOS) sets the position and size of the frame window and of the client window, which is positioned and sized by default inside its parent frame window. When this call is issued, the application appears on the screen as shown in Figure 4-1 on page 4-1, except for the pull-down. Alternatively, the call WinShowWindow (WISHOW) can show or hide a window.

Changes to the Window Procedure

In addition to adding WM_COMMAND processing to the window procedure, the processing of some messages is different from the first example, to allow for the standard window. There is no case statement for the WM_BUTTON1DOWN message, because the frame window has been made active by WinSetWindowPos (WISPOS).

The WM_CREATE Message

When a window is created, WM_CREATE is sent to the window by the function creating it, before the window is made visible. In the processing of the WM_CREATE message, an application usually performs any initialization the window requires. For example, at this point, a real application opens a file or files containing any data that the window requires, and reads the data in.

The sample application uses the WinLoadString (WMLSTR) call to load the character strings IDS_HELLO, IDS_1, IDS_2, and IDS_3 from the stringtable in the resource file into the application variables szHello, sz1, sz2, and sz3. The strcpy function initializes the szString variable with the text in the szHello variable.

If the window requires any window-specific data, it is read into the application at this point. For example, if your window is going to prompt the user to type input into several entry fields, your application could allocate a data structure at this point, and use WinSetWindowULong (WISWUL) to store a pointer to a data structure in a window word. (There is an example of this in "Using Window Words" on page 13-14.) You can use the WinSetWindowUShort (WISWUS) or WinSetWindowULong (WISWUL) calls to set unsigned short or long values in the window words. You can use the WinSetWindowPtr (WISWPT) call to place a pointer to a data object in the window words of a window. You can use the corresponding query calls WinQueryWindowUShort (WIQWUS), WinQueryWindowULong (WIQWUL), and WinQueryWindowPtr (WIWPT) to query the window words contents.

If the application explicitly creates any child windows, it usually does so in the WM_CREATE processing.

The WM_SETFOCUS Message

HELLO2 does not perform any processing as a result of receiving the WM_SETFOCUS message. The only keyboard input to which the application responds is function key input.

A genuine application would test the first part of the second parameter of WM_SETFOCUS. TRUE indicates that an application is receiving the focus. An application typically responds by displaying a text cursor at the position in the window where text is to be typed. FALSE indicates that an application is losing the focus. It typically responds by removing the text cursor from the window.

The WM_COMMAND Message

Pull-down choices generate WM_COMMAND messages when selected. The messages pass to the application's window so that it can execute the function to which those choices relate. Therefore, the window procedure needs a case statement to process the WM_COMMAND message. The WM_COMMAND message has a command code value as part of its data. The command code identifies which application function is requested, matching the definition of the choices, each of which contains a command value as part of its definition. The SHORT1FROMMP macro extracts the command value from the first part of the first message parameter. The command value is used as input to a further set of case statements that are nested inside the processing of the WM_COMMAND message.

Case statements are defined for ID_OPTION1, ID_OPTION2, ID_OPTION3, and ID_EXITPROG that match the identifier values in the resource script file in the example code on page 6-2. Any other codes are ignored and are dealt with by the default case.

ID_OPTION1 , ID_OPTION2, and ID_OPTION3 relate to the choices 1, 2, or 3 from the "Commands" menu. The application processes whichever of the three choices is selected by using the C/2 strcpy function to copy the text in sz1, sz2, or sz3 into the variable szString. The variables hold the three text strings which are alternatives to "Hello." In each case, the application calls WinInvalidateRegion (WINVRG). This call can be used to update only the part of a window that needs updating, as held in the update region. In the

sample, the call specifies the whole window as needing updating. WinInvalidateRegion (WINVRG) causes a WM_PAINT message to be sent to the application. The WM_COMMAND processing contains no drawing functions. All the drawing is done in the WM_PAINT processing.

ID_EXITPROG represents the function of exiting (terminating) the application. The application responds by posting a WM_CLOSE message to itself. ID_RESUME represents the function of continuing with the application (a no-operation instruction). These are related to the "Exit" pull-down choices.

The WM_ERASEBACKGROUND Message

This message is sent by OS/2 if the frame window is redrawn. Returning TRUE causes the window background to be redrawn in the color SYSCLR_WINDOW.

The WM_PAINT Message

The application processes WM_PAINT by calling WinBeginPaint (WIBPNT) to create a presentation space. The Gpi calls GpiSetColor (GSCOL), GpiSetBackColor (GSBCOL), and GpiSetBackMix (GSBMIX) set the color of the foreground and the background of the text string, and define how the background of the text string mixes with the color of the client window.

The application processes WM_COMMAND by setting the contents of a string variable to 1, 2, or 3, depending on the command code value, and calls WinInvalidateRegion (WINVRG), which sends a WM_PAINT message to itself. The WM_PAINT processing uses GpiCharStringAt (GICHAR) to draw the contents of the string variable. In this way, the WM_COMMAND processing is coordinated with the WM_PAINT processing. The first time, the string variable contains "Hello", as initialized in the processing of WM_CREATE. The next time, the string variable contains the text set in the WM_COMMAND processing.

The program terminates itself by posting a WM_QUIT message. This is the same method as when the F3 key is pressed.

The WM_CLOSE Message

WM_CLOSE is generated either when the user selects "Close" from the system menu, or when the application posts a WM_CLOSE in processing the WM_COMMAND message with a command value of ID_EXITPROG. The ID_EXITPROG processing posts a WM_CLOSE so termination is handled at a single point in the window procedure.

When a window procedure receives this message, it might ask the user to confirm closure. When closure is confirmed, the application can close files, release dynamic memory, destroy its windows, and perform other termination actions. In a single-window application like the example, the sensible thing is to terminate the application by posting the WM_QUIT message. If the window is destroyed, the application does not terminate. However, the user cannot terminate the application using the F3 key, because there is no window to receive the keyboard input.

HELLO2 Sample PM Application — Conclusion

The HELLO2 sample application covers much of the structure of a typical PM application. It is easy to see how the application can be extended by more choices, more accelerators, and more functions.

The real function of the application is in the user interface defined by the HELLO2 sample application. This is within the WM_COMMAND processing in the window procedure. If you replace the simple drawing functions with calls to application functions that do real work, a real application begins to take shape.

It is also possible to see that the application can be developed incrementally, one new function at a time. For example, to make the application sensitive to the size of its window, it is necessary only to add processing for the WM_SIZE message sent to the window procedure when the window size changes. You can apply the same principle to other functions.

Chapter 7. Dialog Boxes and Controls

Dialog boxes are windows that an application uses for dialog with the user. Dialog boxes can contain a free-form arrangement of any of the system-provided control windows, plus any required specialized application-provided control windows:

1. Radio buttons
2. Check boxes
3. Pushbuttons
4. List boxes
5. Entry fields
6. Scroll bars
7. Combination boxes.

Control windows are most often used as child windows of a dialog box, but can also be used as descendants of a client window. The dialog box is the most suitable window to use for complex types of user dialog panel, where there may be interrelated sets of fields – for example, a list box, some check boxes, and an entry field.

You define a dialog box and its controls in a *dialog template*. A dialog template defines the identifiers, window classes, class styles, window styles, positions, and text for all the dialog boxes and control windows in a dialog. A dialog template can be either in a resource file, or in memory. This is described in "The Dialog Box Editor" on page 7-21.

To display the dialog box, your application can use one of several calls that use the dialog template. The calls that you use vary according to how you want the user to use the dialog box, and whether you are coding your application in a reentrant or nonreentrant language. An understanding of the reentrant languages support will also help you understand the support for nonreentrant languages.

Message Communication Between a Dialog Box and Its Controls

Like the owner/owned relationship between the WC_FRAME window and its frame controls, a dialog box owns its controls. If a significant event happens to a control window, its window procedure sends or posts a *notification message* to the dialog box. When a dialog box wants to affect or query the state of one of its control windows, it sends a *control message* to the control.

For example, when a user selects a choice in a list box, a WM_CONTROL message with an LN_SELECT notification code is sent to the owner window. If the dialog box wants to find out which choices are selected in the list box, it sends an LM_QUERYSELECTION control message to the list box window.

The notification and control messages vary according to the type of control. They are listed for each control type in the *Control Program Programming Reference* manual.

Dialog Processing in a Reentrant Language

OS/2 handles much of the programming of the user interactions with the dialog box for you. This includes the direct processing of the input to an individual control and the switching from one control to the next – for example, TAB key processing. OS/2 does not handle all the processing that an application may require for a dialog box. So, in an application in a reentrant language, just as a window must have a window procedure to process messages, a dialog box must have an application-defined **dialog procedure**. A dialog procedure is a window procedure for a dialog box. A dialog procedure is similar in form to a window procedure, and processes user interaction with the dialog box.

The dialog procedure processes notification messages, and sends or posts messages to its controls. All control notification messages are sent or posted to the dialog procedure. A dialog procedure often has to process messages that are sent to it. Several windowing calls send messages that call the dialog procedure. As these calls can be in the dialog procedure itself, it must be coded in a programming language that supports reentrant code, as must the application's window procedure.

Also similarly to a window procedure, a dialog procedure has its own special system-provided default procedure, WinDefDlgProc (WCDDP), to which it can pass any messages that it does not want to process.

The dialog procedure handles interactions such as:

- Initialization of the dialog box – establishing current settings of various controls and providing any variable information, such as file names.
- Validation of input to particular controls, in particular entry fields. You may want your application to verify that user input is correct before the dialog box is completed.
- Updating controls where they are affected by the user changing the state of one of the other controls in the dialog box. Selecting one option may make some other option invalid. If so, mark the invalid option so it can be seen as invalid.
- Connecting the status of the controls to internal program variables. Your application usually is not interested in notification messages until completion of user interaction with the dialog box. Your application's dialog procedure usually then registers the new state of the controls in internal application variables.

There are examples of dialog procedures in Chapter 9, "DIALOG1 Dialog Box Sample Application" and Chapter 10, "The DIALOG2 Sample Application."

Dialog Processing in a Nonreentrant Language

Dialog boxes of an application in a nonreentrant language cannot have dialog procedures, because a nonreentrant language does not support the processing of sent messages.

Messages that are posted to a dialog box are processed by a dialog routine in your application. Similarly to a dialog procedure, it passes all messages that it does not explicitly process to a system-provided default procedure.

Messages that are sent to a dialog box are processed by the language-support dialog procedure. Like the language-support window procedure, for most messages it replies in a default manner. For a subset of messages, the language-support dialog procedure first takes default action, then posts a renamed version of the message to the application's input queue for processing by the application's dialog routine. This subset of messages is the same as that for the language-support window procedure.

The language-support dialog procedure can also be used by a C/2 or Macro Assembler/2 program that does not have a dialog procedure.

Modeless and Modal Dialogs

In a reentrant or nonreentrant language, once a dialog box has been created, it can be used in a *modeless* dialog, or a *modal* dialog, depending on the kind of user interaction the application requires.

Modeless Dialog

In a modeless dialog, the user is free to interact with the dialog box and any other object on the screen. A modeless dialog can be used, for example, to let the user see and change data display options that relate to data in the main window, while, at the same time, letting the user view and even change the data itself.

Modeless Dialog in a Reentrant Language

The basic structure of a simple reentrant PM application with a single modeless dialog box is as follows:

```
MainProc
  Initialize
  Create an input queue
  Register the window procedure (WinProc)
  Create a window
  Loop until a termination message is received
     Get the next posted message
     Dispatch it to window or dialog procedure
Terminate


WinProc
  Process messages for which procedure
  has to take specific action.
  When an event occurs that requires
  a dialog box, call a function to
  load and display dialog box whose
  input will be processed by DlgProc.
  Pass everything else to system-provided
  default window procedure WinDefWindowProc
Return
```

```
DlgProc
  Process messages for which procedure
  has to take specific action.
  Pass everything else to system-provided
  default dialog procedure WinDefDlgProc
Return
```

A modeless dialog is obtained as follows:

- When the dialog template is in a resource file, use the WinLoadDlg (WCLD) call, followed by WinShowWindow (WISHOW) to make the dialog box window visible.
- When the dialog template is in memory, use the WinCreateDlg (WCCRD) call, followed by WinShowWindow (WISHOW) to make the dialog box window visible.

Messages are passed to the dialog box window in the usual way – they are received using WinGetMsg (WMGET) and sent using WinDispatchMsg (WMDISP) in the main procedure of the application.

Completion is determined by the dialog procedure. When the dialog is complete, the dialog procedure can call WinDismissDlg (WCDELD).

WinDismissDlg merely hides a modeless dialog box, and control returns to the caller of WinDlgBox (WCLMD) or WinProcessDlg (WCPD). The dialog box window still exists at this point, which allows the state of its various controls to be examined using WinQueryDlgItemText (WMQDIT).

When the dialog box is complete, as determined by the dialog procedure, the dialog box subsequently can be destroyed by a call to WinDestroyWindow (WIDEL), or reused by calling WinDlgBox (WCLMD) with the same dialog identifier.

Modeless Dialog in a Nonreentrant Language

The basic structure of a simple nonreentrant PM application with a single modeless dialog box is as follows:

```
MainProc
  Initialize
  Create an input queue
  Register null window class procedure
  Create a window
  Loop until a termination message is received
    Get the next posted message
    If destination window is my window,
      call MyRoutine
    If destination window is my dialog box,
      call MyDlgRoutine
    Else dispatch to appropriate window procedure.
Terminate

MyRoutine
  Process posted messages for which routine
  has to take specific action.
  When an event occurs that requires
  a dialog box, call a function to
  load and display dialog box.
  Pass everything else to system-provided
  default window procedure WMDWP
End

MyDlgRoutine
  Process posted messages for which procedure
  has to take specific action.
  Pass everything else to system-provided
  default dialog procedure WCDDP
End
```

A modeless dialog is obtained as follows:

- When the dialog template is in a resource file, use the WinLoadDlg (WCLD) call, followed by WinShowWindow (WISHOW) to make the dialog box window visible.
- When the dialog template is in memory, use the WinCreateDlg (WCCRD) call, followed by WinShowWindow (WISHOW) to make the dialog box window visible.

In both cases, the call should specify a null dialog procedure. This tells the system to use the language-support dialog procedure to intercept messages sent to the dialog box.

The main message-processing loop of the application deals with all posted messages for both the main window and any dialog boxes, by checking the destination of the window message, and passing the message on to its destination.

Modal Dialog

In a modal dialog, the user is prevented from interacting with certain objects on the screen, depending on how you specify the modality.

When you create the dialog template, you can specify it as a *system-modal* dialog. In this case, when the dialog box is displayed, the user is prevented from interacting with *everything* else in the system except the dialog box. This includes the other parts of the application that displayed the dialog box, and *every other application in the system.* The occasions when you need the system-modal facility are rare. One possible use could be to display information when an application is in danger of bringing the system down.

If a dialog template is not specified as system-modal, the level of modality can be controlled by the application. An *application-modal* dialog box prevents user interaction with its owner *and all the descendants of its owner.*

To conform with the Common User Access definition of user interaction with a pop-up window, an application-modal dialog box must be created with the desktop as its parent, and the frame as its owner. The parent and owner are specified on either the WinLoadDlg (WCLD), WinDlgBox (WCLMD), or WinCreateDlg (WCCRD) calls. The specification of the desktop as its parent means that the dialog box is clipped to the desktop, and so can overlap the frame. The specification of the frame as owner means that the user is prevented from interacting with the frame and any of the descendants of the frame. The user can interact with the dialog box, is restricted from using the application that displayed it, but can still interact with any other application in the system.

The fact that interaction is prevented with the owner of a dialog box and all the descendants of its owner means that you must take great care if you specify anything other than the Common-User-Access-conforming arrangement of the desktop as the parent and the frame as the owner. For example, consider the following three cases of non-conformance:

- If the desktop is the parent and the client window is the owner, the user is prevented from interacting with the client window and all the descendants of the client. The user can still interact with the dialog box because its parent is the desktop. The user can also interact with the frame, the frame controls, and other applications.
- If the frame is the parent *and* the owner, the dialog box is clipped to the frame. The user is prevented from interacting with the owner of the dialog box, which is the frame, and all the descendants of the frame window. These include the client window and the *dialog box itself.* The user can still interact with other applications.
- If the client window is the parent *and* the owner, the dialog box is clipped to the client window. The user is prevented from interacting with the owner of the dialog box, which is the client window, and all the descendants of the client window. Again, the descendants include the *dialog box itself.* The user can still interact with other applications.

Modal dialog boxes are useful where the user must supply some information before a particular function can proceed. It is good user-interface design to allow the user to abandon the interaction, if necessary, by providing a **Cancel** option somewhere in the dialog box, in addition to the usual **Enter** option that allows the user to submit the panel (that is, indicate that it is complete). In a modal dialog, once the dialog box is displayed, the user must normally complete the dialog before continuing to interact with the remainder of the application.

Modal Dialog in a Reentrant Language

The basic structure of a simple reentrant PM application with a single modal dialog box is as follows:

```
MainProc
  Initialize
  Create an input queue
  Register the window procedure (WinProc)
  Create a window
  Loop until a termination message is received
     Get the next posted message
     Dispatch it to the window procedure
Terminate

WinProc
  Process messages for which procedure
  has to take specific action.
  When an event occurs that requires
  a dialog box, call a function to
  load and display dialog box.
  Control passes to called function,
  that uses DlgProc to process input.
  Pass everything else to
  system-provided WinDefWinProc.
Return

DlgProc
  Process messages for which procedure
  has to take specific action.
  Pass everything else to
  system-provided WinDefDlgProc.
Return
```

A modal dialog is obtained as follows:

- When the dialog template is in a resource file, use either the WinLoadDlg (WCLD) and WinProcessDlg (WCPD) calls, or the WinDlgBox (WCLMD) call, which is a functional combination of the WinLoadDlg and WinProcessDlg calls.
- When the dialog template is in memory, use the WinCreateDlg (WCCRD) and WinProcessDlg (WCPD) calls.

WinLoadDlg (WCLD) creates the dialog box invisible, unless it has the WS_VISIBLE window style. It is usually better to create the dialog box invisible, as subsequent user input caused by the user typing ahead can obviate the need to display the dialog box.

WinProcessDlg disables the owner of the dialog, and does not return until the dialog is completed. During this time, WinProcessDlg does the job of a message processing loop by getting and dispatching messages from the application queue to the appropriate control window procedure, or to the dialog procedure. WinProcessDlg does not display the dialog box while there are still WM_CHAR messages in the application queue, but allows these messages to be processed first. WinProcessDlg gets and dispatches messages until a message causes the dialog procedure to complete, or until all the messages from the application queue are processed.

The dialog procedure related to the dialog box receives various messages that pass to the dialog box, and can act on them as required.

Completion is determined by the dialog procedure. When the dialog is complete, the dialog procedure calls WinDismissDlg (WCDELD). When a modal dialog box has been dismissed by WinDismissDlg, WinDlgBox or WinProcessDlg destroys the dialog box, so that information cannot subsequently be obtained from it.

Modal Dialog in a Nonreentrant Language

The basic structure of a simple nonreentrant PM application with a single modal dialog box is as follows:

```
MainProc
  Initialize
  Create an input queue
  Register the window procedure (WinProc)
  Create a window
  Loop until a termination message is received
    Get the next posted message
    If destination window is my window,
      call MyRoutine
    Else dispatch to appropriate window procedure
Terminate


MyRoutine
  Process posted messages for which routine
  has to take specific action.
  When an event occurs that requires
  a dialog box, call a function to
  load and display dialog box,
  and call MyDlgRoutine to process input.
  Pass everything else to system-provided
  default window procedure WMDWP
End

MyDlgRoutine
  Loop until a termination message is received
    Get the next posted dialog message
    Process posted messages for which procedure
    has to take specific action
    Dispatch everything else to system-provided
    default dialog procedure
End
```

A modal dialog is obtained as follows:

- When the dialog template is in a resource file, use the WinLoadDlg (WCLD) call to create the dialog box.
- When the dialog template is in memory, use the WinCreateDlg (WCCRD) call to create the dialog box.

In both cases, the call should specify a null dialog procedure. This tells the system to use the language-support dialog procedure to intercept messages sent to the dialog box.

Observe that, unlike a modeless dialog, you cannot use the main message-processing loop to deal with messages posted to the dialog box. A modal dialog implies a message-processing loop that processes dialog messages separately from the main message-processing loop. Use the WinGetDlgMsg (WCGETD) call at the start of the dialog routine to get from the input queue messages that are posted to the dialog box. WinGetDlgMsg dispatches messages for windows other than the dialog box. WinGetDlgMsg returns TRUE if the dialog box exists and a message for it is returned.

WinGetDlgMsg (WCGETD) disables the owner of the dialog box. The owner is usually the main window of the application, so the user is prevented from interacting with anything other than the dialog box. The owner is reenabled when WinDismissDlg (WCDELD) is called.

In your application, the dialog routine processes the posted messages in which it is interested, and dispatches unprocessed messages to the dialog box. Because dispatching a message sends it, the message is intercepted by the language-support dialog procedure, which performs a default action.

Passing Control Data to a Newly-Created Dialog Box

Through the WinDlgBox (WCLMD), WinLoadDlg (WCLD), and WinCreateDlg (WCCRD) calls, a window procedure calls a dialog procedure indirectly. A method is therefore required to pass data from one to the other.

"Passing Control Data and Presentation Parameters to a Newly-Created Window" on page 5-6 describes how the WM_CREATE message is sent to a window procedure before the window is displayed. WM_CREATE passes control data to a window procedure, giving the procedure the opportunity to initialize the window.

In a similar way, WinDlgBox (WCLMD), WinLoadDlg (WCLD), and WinCreateDlg (WCCRD) send the WM_INITDLG message to the dialog procedure before the dialog box is displayed. The second parameter of WM_INITDLG can contain an application-defined structure that is shared by the calling procedure and the dialog procedure, and can be used to pass data between the two windows. The application-defined structure is specified in the CreateParams parameter of WinDlgBox, WinLoadDlg, or WinCreateDlg.

A dialog box without a dialog procedure does not receive the WM_INITDLG message.

Sending and Posting a Control Message to a Control Window

To post a control message to a control in a dialog box, you can use WinPostMsg (WMPOST), as you would for any window. Given the identifier, WinWindowFromID returns the window handle for WinPostMsg.

You can use the WinSendDlgItemMsg (WCSDIM) call to send a message to a child window of any window, but you usually use it to send a message to a control window that is a child of a dialog box. (WinSendDlgItemMsg is equivalent to a WinSendMsg call with the window handle returned by WinWindowFromID as its first parameter.)

Setting Text in a Control Window

Similarly to a frame window, a dialog box and its control windows can each have window text. Given the handle of a dialog box or a control window that has a dialog procedure or window procedure, you can use WinSetWindowText (WISTXT) and WinQueryWindowText (WIQTXT) to set and query its window text.

In addition, there is a set of calls that you can use to set and query the text of a child window of any window, but that you usually use to set or query the text of a control window that is a child of a dialog box. The WinSetDlgItemText (WMSDIT)

call sets the text of the child window with the specified identifier and parent window handle.

Control Windows

Controls are the user-interface components that allow users to select choices and enter information. Like other aspects of the interface, controls provide both operational and visual consistency.

Each of the system-provided control window classes has a unique visual appearance and is designed to handle a specific kind of user interaction:

- Making choices, using *selection fields* as follows:

 Single choice – radio buttons
 Multiple choice – check boxes
 Immediate Action – pushbuttons.

- Scrolling selection fields – list boxes
- Typing in text – entry fields
- Scrolling panel contents – scroll bars.

For displaying error messages to the user, there is also a special type of dialog box, called a *message box*, described in "Displaying Errors – Message Boxes" on page 7-19.

Like other system-provided window classes, control windows respond to standard window management messages and functions, such as WinShowWindow (WISHOW) and WinEnableWindow (WISENA).

All the above control window classes can be children of a dialog box, or descendants (usually children) of the client window. The control window classes are described in the sections that follow.

Single Choice – Radio Buttons

When you want your application to give the user a fixed list of choices where only one of the choices can be selected at a time, you can use a group of *radio buttons.*

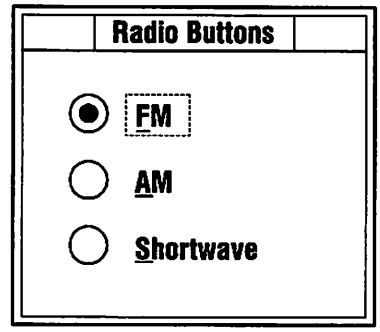


Figure 7-1. Radio Buttons in a Dialog Box

A radio button consists of a small circle followed by a text string that describes the choice. A set of radio buttons is a single selection field that is used to present a fixed set of mutually exclusive choices. When the user clicks on one radio button in the group, the previously selected one is deselected. The user can switch the selection status of the radio button by selecting the text or the button using either the mouse or the keyboard. Alternatively, the user can type the mnemonic character that is indicated in the text string by an underscore. When the user selects a radio button, OS/2 sends a WM_CONTROL message to the owner of the radio button. The message contains the identifier of the radio button and a BN_CLICKED notification code.

The circle that represents a radio button can be empty, to indicate that it is not selected, or can be filled with black, to indicate that it is selected. When the dialog box is displayed, the current selection (either an initial setting or the most recent selection) is shown. Radio buttons are WC_BUTTON class windows, with button styles of either BS_RADIOBUTTON or BS_AUTORADIOBUTTON. These styles have the following effect:

BS_RADIOBUTTON
Your application must respond to the WM_CONTROL message containing the BN_CLICKED notification code by sending a BM_SETCHECK control message to the control.

If the button was in an unchecked state before the WM_CONTROL message was received, the first parameter of BM_SETCHECK must be set to TRUE. The button responds to the message by painting itself black, to indicate that it is selected. Your application must also clear the previously selected button, by sending it a BM_SETCHECK message with the first parameter set to FALSE. The button

responds to the message by painting itself in the background color, to indicate that it is not selected.

BS_AUTORADIOBUTTON
Your application can ignore the WM_CONTROL message containing the BN_CLICKED notification code, as OS/2 automatically fills the selected radio button and clears the previous selection.

Multiple Choice – Check Boxes

When you want your application to give the user a fixed list of choices, where the user can select more than one option, you can use a group of *check boxes.*

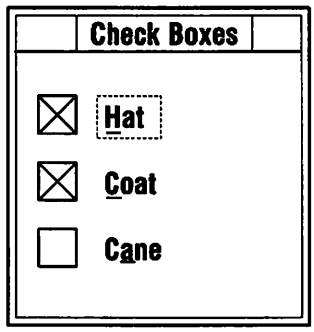


Figure 7-2. Check Boxes in a Dialog Box

A check box consists of a small square followed by a text string that describes the choice. The user can switch the selection status of a check box by selecting the text or the box using either the mouse or the keyboard. Alternatively, the user can type the mnemonic character that is indicated in the text string by an underscore.

A check box acts like a switch that can be toggled on/off by the user. Selection of a specific check box does not affect any other selections. The check boxes accumulate the selections, unlike a radio button field where the choices are mutually exclusive. For example, selecting "Hat" does not deselect "Coat".

Like a radio button, when the user clicks on a check box, OS/2 sends a WM_CONTROL message to its owner containing the identifier of the check box and a BN_CLICKED notification code.

A check box can be empty, indicating that the choice has not been selected, or it can have a cross inside it, indicating that the choice has been selected.

Check boxes are WC_BUTTON class windows, with button styles of either BS_CHECKBOX or BS_AUTOCHECKBOX. Like radio buttons, the button styles specify whether your application will manage the check box, or let OS/2 manage it completely. Your application can check or uncheck BS_CHECKBOX check boxes by sending them BM_SETCHECK messages.

When the dialog box is displayed, the current selection (either an initial setting or the most recent selection) should be shown.

Immediate Action – Pushbuttons

Where you want to give the user the ability to perform an immediate action, you use a *pushbutton.*

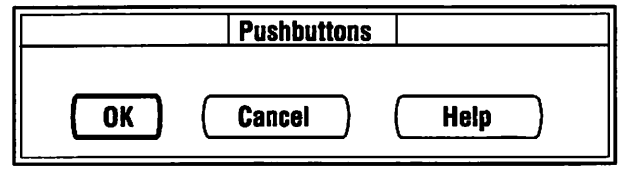


Figure 7-3. Pushbuttons in a Dialog Box

A pushbutton consists of a text string, typically a simple instruction or short item of information, enclosed by a rectangle with rounded corners. The user pushes the button by selecting it with the mouse or the keyboard.

When the user clicks on a pushbutton, OS/2 sends a WM_COMMAND message to the owner of the pushbutton. The message contains a command value that indicates the action selected, and a source code of CMDSRC_PUSHBUTTON. A pushbutton that has been selected is highlighted.

Pushbuttons are WC_BUTTON class windows, with a button style of BS_PUSHBUTTON.

To conform with Common User Access, radio buttons and check boxes should be contained in a dialog box that also contains pushbuttons that let the user enter or cancel the selection. In this case, an application would query the status of the radio buttons or check boxes only when the user clicks on the enter pushbutton.

The pushbuttons should appear horizontally in the lower portion of the dialog. If this does not adequately support the particular dialog box layout, they may also appear vertically in the right portion of the dialog. There is a standard set of pushbuttons for use in dialog boxes; for details of these refer to the Common User Access publications.

Other Button Styles

BS_3STATE
When used *instead* of BS_CHECKBOX, produces a check box whose appearance can be set to any of the three states halftoned, checked, or unchecked.

BS_AUTO3STATE
When used instead of BS_CHECKBOX or BS_3STATE, produces a three-state check box that, when selected, automatically switches between the three states.

BS_USERBUTTON
This is a button whose appearance is defined by your application. Your application receives a BN_PAINT message when this needs painting.

BS_NOPOINTERFOCUS
This style can be ORed with BS_RADIOBUTTON, BS_AUTORADIOBUTTON, BS_CHECKBOX, BS_AUTOCHECKBOX, BS_PUSHBUTTON, BS_3STATE, BS_AUTO3STATE, or BS_USERBUTTON styles. Buttons with this style do not set the focus to themselves when clicked. This allows you, for example, to keep the text cursor on an entry field when text input is required, rather than the focus moving to the button.

BS_NOCURSORSELECT
This style can be ORed with button style BS_AUTORADIOBUTTON. The result is that the button is not automatically selected if it receives the selection cursor by use of the cursor-movement keys.

BS_DEFAULT
This style can be ORed with the BS_PUSHBUTTON or BS_USERBUTTON styles. With this style, a pushbutton has a thick border. A pushbutton operates in the same way, and can also be selected by the Enter and Return keys.

The following styles can be ORed with the BS_PUSHBUTTON style:

BS_HELP
Posts a WM_HELP message instead of a WM_COMMAND message.

BS_SYSCOMMAND
Posts a WM_SYSCOMMAND message instead of a WM_COMMAND message.

BS_NOBORDER
The button is displayed without a border.

Receiving Button Notification Messages

As mentioned above, the selection state of a WC_BUTTON window is sent to the application in a WM_CONTROL message. The WM_CONTROL message contains a notification code that indicates the nature of the user's interaction. The selection state of WC_BUTTON class window can also be inspected by the owner window sending a BM_QUERYCHECK control message. To allow the user to make a selection without entering it to the application, your application should wait until the user uses a pushbutton to enter the selection, before acting on the selection.

Button Notification Codes

BN_CLICKED
The button has been clicked

BN_DBLCLICKED
The button has been double-clicked

BN_PAINT
The button requires painting. The interpretation of this is affected by the button draw state codes in the second message parameter:

BDS_HILITED
The highlighted state of the button requires painting

BDS_DISABLED
The disabled state of the button requires painting

BDS_DEFAULT
The appearance is application-defined.

Sending and Posting Button Control Messages

You can send messages with a BM_ prefix, to control buttons:

Button Control Messages

BM_CLICK
Has the same effect as a user clicking a push-button

BM_QUERYCHECK
Returns the checked state of a button

BM_QUERYCHECKINDEX
Returns the zero-based index offset of a checked radio button

BM_QUERYHILITE
Returns the highlight state of a button

BM_SETCHECK
Sets the checked state of a button

BM_SETDEFAULT
Sets the default state of a button

BM_SETHILITE
Sets the highlight state of a button.

Scrollable Selection Fields — List Boxes

An application uses a list box when it requires a list of selectable fields that is too large to fit into the screen area available, or a list of choices that can change dynamically.

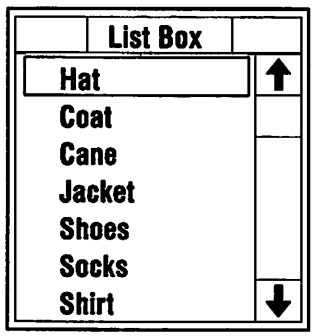


Figure 7-4. A List Box in a Dialog Box

A list box is a window that displays several text choices at a time. The user can select choices in the usual way with the mouse or the keyboard.

A list box contains a scroll bar that is enabled by the system when the total list of choices in the list box exceeds the vertical space available for display. The scroll bar indicates to the user which portion of the list is displayed and also lets the user scroll through the complete list, as required. To scroll through the list, the user can use the cursor control keys, or the Page Up and Page Down keys, or manipulate the scroll bar with the mouse.

The length of the text string in a choice can be such that its width is greater than the width of the list box. In this case, you should supply a horizontal scroll bar. The slider position shows the position and size of the list box width relative to the widest data choice in the list. (The widest choice includes two spaces.)

When the user scrolls as far as is possible in one direction, the scroll bar arrow that points in that direction is grayed and therefore cannot be selected.

List boxes are WC_LISTBOX class windows that can have the following LS_ list-box styles:

LS_HORZSCROLL
Includes a horizontal scroll bar.

LS_MULTIPLESEL
Allows multiple selections at one time. The default is single selection.

LS_NOADJUSTPOS
Draws the list box at the specified size.

LS_NOVERTSCROLL
Removes the vertical scrollbar.

LS_OWNERDRAW
Allows one or more list box items to be drawn by the owner. Typically, these are bit maps rather than text strings.

Note: There is no list box style provided in OS/2 1.2 to create the Extended Selection interaction described in Common User Access.

Maximum Number of Items in a List Box

The maximum number of items allowed in a list box depends on the amount of storage allocated to the list box. Storage is allocated dynamically by the Heap Manager (see "The Heap Manager" on page 13-22) and list boxes use the message-queue heap, which also stores all the menu items and entry fields for your application.

Note: The Heap Manager is not supported in COBOL and FORTRAN.

However, you can increase the amount of storage available for a list box (up to a maximum of 64KB[1]) by intercepting the WM_CONTROLHEAP message that is sent when a list box control is created. You can then use the DosAllocSeg call to allocate a separate segment, followed by the WinCreateHeap call to create a heap within the segment. Return the heap handle to the list box control in response to the WM_CONTROLHEAP message.

Table 7-1 gives a guide to the number of fixed-size items that can be included in a list box that has a 64KB[1] heap. These guidelines assume that the list box items are added consecutively.

Table 7-1. Maximum Numbers of Items in a List Box. Guidelines for a 64KB[1] Heap.

Item Length (bytes)	Number of Items Allowed Is At Least
less than 5	3030
5 to 8	2840
9 or 10	2390
11 to 20	1600
21 to 30	1325
31 to 40	1125

Receiving List Box Notification Messages

When the user interacts with a list box, the list box sends a WM_CONTROL notification message to its owner window. Examples of user interaction are the selection of a choice, scrolling of the list box contents, and pressing the Enter key. The notification message includes a notification code that indicates which action the user has performed.

List Box Notification Codes

LN_ENTER
The Enter or Return key has been pressed while the list box has the input focus, or the list box has been double-clicked

LN_KILLFOCUS
The list box is losing the input focus

LN_SCROLL
The list box is about to scroll

LN_SETFOCUS
The list box is receiving the input focus

LN_SELECT
An item is being selected or deselected.

Sending and Posting List Box Control Messages

Your application can modify or query the status, content, and other information about list box items by sending or posting messages that have an LM_ prefix. For example, selected choices can be found by using the LM_GETSELECTION message, and you can add items to or delete items from the list, and change their text, using various LM_* messages. The enabled/disabled state of the vertical scroll bar changes dynamically as items are added to or deleted from the list box.

List Item Index

The items in a list box have a zero-based index that you can use in some of the LM_ messages to specify the offset position of an item. To use the index, you specify either your own offset from zero, or a system-provided LIT_ index value. The first item in a list box has the item index LIT_FIRST.

List Box Control Messages

LM_DELETEALL
Deletes all the items

LM_DELETEITEM
Deletes an index item

LM_INSERTITEM
Inserts an item at specified index or:

LIT_END
At the end of the list

LIT_SORTASCENDING
In ascending order

LIT_SORTDESCENDING
In descending order.

LM_QUERYITEMCOUNT
Returns a count of the number of items

[1] 1KB = 1024 bytes.

LM_QUERYITEMHANDLE
Returns the handle of an indexed item

LM_QUERYITEMTEXT
Loads the text of an indexed item into a specified buffer

LM_QUERYITEMTEXTLENGTH
Returns the length of the text of an indexed item

LM_QUERYSELECTION
Returns the index offset of a selected item that follows a specified offset

LM_QUERYTOPINDEX
Returns the index offset of the item currently at the top of the list box

LM_SEARCHSTRING
Returns the index offset of the item that matches (or partly matches, as specified by a command value) a specified string

LM_SELECTITEM
Selects or deselects the indexed item

LM_SETITEMHANDLE
Sets the handle of an indexed item

LM_SETITEMHEIGHT
Sets the height of the items

LM_SETITEMTEXT
Copies text from a specified buffer into an indexed item

LM_SETTOPINDEX
Scrolls an indexed item to the top of the list box.

Example Code – Sending a List Box Control Message

Here is an example of WinSendDlgItemMsg (WCSDIM), that sends an LM_INSERTITEM to insert an item as the last item (given by LIT_END) in a list box called LISTBOX:

```
WinSendDlgItemMsg( hwndDlg,
                   LISTBOX,
                   LM_INSERTITEM,
                   MPFROM2SHORT( LIT_END, 0 ),
                   MPFROMP( szBuffer ));
```

Text Entry – Entry Fields

When an application requires text input from the user, it creates an *entry field.*



Figure 7-5. Entry Field Example

An entry field consists of a rectangular window into which the user can type a line, or lines, of text. Two types of text entry windows are defined:

Single line entry For entering a single line of text.

Multiline entry For entering several lines of text or importing and exporting blocks of text.

An entry field is shown in Figure 7-5.

When the entry field has the *input focus*, a text cursor is displayed at the insertion or replacement point. The Insert key switches alternately between insertion and replacement. The user can move the cursor using the keyboard arrow keys, and can also use the Delete key to delete text from the field. The user can press-and-hold the pointer, and drag it across text, to select it. Selected text is shown in reverse color. The user can also use the cut (Shift + Delete), copy (Ctrl + Insert), and paste (Shift + Insert) methods to move selected text to and from the clipboard. (For a description of the clipboard, see Chapter 28, "User-Driven Data Exchange – The Clipboard.")

Single Line Entry Fields

When your application requires the user to enter only a single line of text you use a single line entry field. Single line entry fields are WC_ENTRYFIELD class windows, and can have ES_ entryfield styles:

ES_LEFT
Text is left-justified. This is the default, if ES_RIGHT or ES_CENTER are not specified.

ES_RIGHT
Text is right-justified.

ES_CENTER
Text is centered.

ES_MARGIN
Draws a frame, with a margin, around the text.

ES_PICTUREMASK
The control data is interpreted as a picture mask. The mask consists of two pattern characters, L and blank. L indicates that the character in the initial text is protected and cannot be overwritten. Blank indicates unprotected data.

ES_READONLY
The text field is read only.

ES_UNREADABLE
Creates a field which is not displayed on the screen.

Note: This style can only be set at field definition time and cannot be changed once it is specified.

Instead of the single-byte character set (SBCS) representation used for Latin text, some Asian countries use code pages that consist of double-byte character set (DBCS) characters, where each character is represented by a two-byte code. The DBCS code pages allow single-byte, double-byte, or mixed (single-byte and double-byte) data. For these countries, OS/2 provides the following additional styles:

ES_SBCS
Text is purely single-byte.

ES_DBCS
Text is purely double byte.

ES_MIXED
Text is a mixture of SBCS and DBCS characters.

ES_ANY
Text is a mixture of SBCS and DBCS characters. This is the default.

Note: If the queue code page is ASCII and the data is subsequently converted to an EBCDIC code page, the length of the EBCDIC field, contains shift-out and shift-in control characters around DBCS characters. This causes the EBCDIC data to be longer than the original ASCII field. Coding ES_MIXED protects the target field from overrun in this situation.

The text string typed into a single line entry field can be larger than the width of the rectangular window. When the user keys in more text than can fit into the visible part of the field, the text scrolls automatically. When the user attempts to overtype the last character position, or when the user tries to insert text into a field that is already full, the system issues a warning beep, except in the case of a one-character field. In this particular case, the character is replaced whether in insert mode or replace mode.

A field, or part of a field can be protected from being overtyped or erased by input from the keyboard. Use the **ES_READONLY** to protect the whole field and use the **ES_PICTUREMASK** style to protect part of a field.

Receiving Single Line Entry Field Notification Messages

The application receives WM_CONTROL notification messages when the user changes or scrolls the text in the entry field.

Single Line Entry Field Notification Codes

EN_SETFOCUS
Receiving the focus

EN_KILLFOCUS
Losing the focus

EN_CHANGE
The displayed content has changed

EN_SCROLL
The field is about to scroll horizontally

EN_MEMERROR
The system cannot allocate the storage to accommodate the text length implied by the EM_SETTEXTLIMIT (see below)

EN_OVERFLOW
The input exceeds the current text limit.

Sending and Posting Single Line Entry Field Control Messages

Single Line Entry Field Control Messages

EM_CLEAR
Delete selected text from the field

EM_COPY
Copy selected text from the field to the clipboard, in CF_TEXT format

EM_CUT
Send selected text from the field to the clipboard, and delete it from the field

EM_PASTE
Copy selected text from the clipboard into the field

EM_QUERYCHANGED
Return whether the text has changed

EM_QUERYREADONLY
Return whether the field is read only

EM_QUERYFIRSTCHAR
Returns the zero-based offset of the first displayed character in the field

EM_QUERYSEL
Return the zero-based offsets of the start and end of selected text

EM_SETFIRSTCHAR
Specify the zero-based offset of the first displayed character

EM_SETINSERTMODE
Set or clear insert mode

EM_SETREADONLY
Set or clear read only mode

EM_SETSEL
Set the zero-based offsets of the start and end of the selected text

EM_SETTEXTLIMIT
Set the maximum number of bytes that the entry field can contain. The default maximum length of an entry field is 32 bytes. You cannot increase the size using the Dialog Box Editor. Instead, your application must send the entry field this message.

Your application can also use the WM_QUERYWINDOWPARAMS control message to find out the text currently in the entry field – the text displayed is the window text of the entry field control window. You can also use the WinQueryWindowText (WIQTXT) call.

Multiline Entry Fields

When your application requires the user to enter more than a single line of text, you use a multiline entry field. A multiline entry field is a WC_MLE class window and can have MLS_ entry field styles:

MLS_READONLY
The initial state of the window is read-only

MLS_BORDER
A thin border is drawn round the entry window

MLS_WORDWRAP
The initial state of the window is with word-wrap on

MLS_HSCROLL
The window contains a horizontal scroll bar

MLS_VSCROLL
The window contains a vertical scroll bar

MLS_IGNORETAB
Causes the multiline edit control to ignore Tab key strokes and pass them on to the application.

A multiline entry field uses the dialog code DLGC_MLE to respond to dialog query messages.

Multiline entry fields give the user the choice of deciding where the line breaks will be, or allowing the system to create a new line when the end of the visible field is reached. This facility is known as **word-wrap**. When the user attempts to overtype the last character position, or when the user tries to insert text into a field that is already full, the system issues a warning beep. Use **Read only** mode to protect an entry field from overtyping or erasure by keyboard input. Data can be imported and exported to and from a multiline entry field. The format for the data is determined by the **MLM_FORMAT**. message. The fonts and colors for use in the multiline entry field can be chosen from those available on the system. The multiline entry field has the ability to undo the last operation which changed the contents of the text. If an undo is not possible, a notification is sent to the application. If large changes are being made to the multiline entry field, it may be advisable to use the **MLM_DISABLEREFRESH** message to prevent multiple updates of the screen while the data is being changed.

When a WM_CREATE message is sent it can point to an MLECTLDATA structure which defines the format of the multiline entry field. If this pointer id not included the following defaults are used:

CCHTEXT
Maximum text allowed in MLE. Default is value given in MLM_SETTEXTLIMIT.

IPTANCHOR
Beginning point of selection. Default is 0.

IPTCURSOR
End point of selection field. Default is 0.

XFORMATSIZE and YFORMATSIZE
The dimensions in pixels of the formatting rectangle. Default is the window size in both dimensions.

AFFORMATFLAGS
Govern the interpretation of xFormatSize and yFormatSize fields. Default is unlimited in both directions and a varying size to match the window size.

USFORMAT
The initial import/export format. Default is MLE_CFTEXT.

Receiving Multiline Entry Field Notification Messages

The application receives WM_CONTROL notification messages when the user changes or scrolls the text in the entry field.

Multiline Entry Field Notification Codes

MLN_TEXTOVERFLOW
The input exceeds the current text limit.

MLN_PIXHORZOVERFLOW
The input exceeds the horizontal limit of the entry field format rectangle.

MLN_PIXVERTOVERFLOW
The input exceeds the vertical limit of the entry field format rectangle.

MLN_OVERFLOW
An entry field control message has changed the text limit or the entry field format rectangle such that the input will no longer fit.

MLN_HSCROLL
Horizontal scroll is about to take place.

MLN_VSCROLL
Vertical scroll is about to take place.

MLN_CHANGE
Text change has occurred.

MLN_UNDOOVERFLOW
Text exceeds the undo limit. Undo cannot be performed.

MLN_CLPBDFAIL
Clipboard operation failed.

MLN_MEMERROR
The system cannot allocate the storage required by the action.

MLN_SETFOCUS
Window has received the focus.

MLN_KILLFOCUS
The window has lost the focus.

MLN_MARGIN
A mouse event has occurred in the margins of the entry field. If the owning window does not process the notification the event is treated as if it occurred on the closest point in the text.

MLN_SEARCHPAUSE
A search message is being processed.

Sending and Posting Multiline Entry Field Control Messages

MLM_SETTEXTLIMIT
Set a limit to the number of bytes of text the entry field can hold. The default is unbounded.

MLM_QUERYTEXTLIMIT
Return the current limit.

MLM_SETFORMATRECT
Set the horizontal and vertical limits for word-wrap.

The afFlags parameters for this message define how the rectangle is to be applied.

MLM_QUERYFORMATRECT
Return the horizontal and vertical limits for word-wrap.

MLM_SETWRAP
Set or clear word-wrap mode and updates the screen.

MLM_QUERYWRAP
Return the status of word-wrap mode.

MLM_SETTABSTOP
Set the intervals for tab-stops in pels.

MLM_QUERYTABSTOP
Return the tab-stop intervals in pels.

MLM_SETREADONLY
Set or clear read only mode.

MLM_QUERYREADONLY
Return the status of read only mode.

MLM_INSERT
Inserts the text string at the current selection, replacing the current text.

MLM_QUERYSELTEXT
Copies the currently selected text into a buffer.

MLM_SETCHANGED
Set the state of the text changed flag.

MLM_QUERYCHANGED
Return the status of the text changed flag.

MLM_QUERYLINECOUNT
Return the number of lines in the entry field.

MLM_CHARFROMLINE
Return the first insertion point on a given line.

MLM_LINEFROMCHAR
Return the line number for a given insertion point.

MLM_FORMAT
Set the format for importing and exporting text.

The following formats are available:

MLE_CFTEXT
Clipboard text format. This is the default.

MLE_NOTRANS
Guarantees that ant text imported into the MLE can be recovered in the same form by exporting.

MLE_WINFMT
On import, recognizez CR LF as hard line breaks and ignores CR CR LF. On export, uses CR LF for hard line braks and CR CR LF for soft line breaks caused by word-wrapping.

MLM_QUERYLINELENGTH
Return the number of bytes between an insertion point and the end of the line without current formatting.

MLM_QUERYFORMATLINELENGTH
Return the number of bytes between an insertion point and the end of the formatted line.

MLM_QUERYTEXTLENGTH
Return the number of text characters, in the entry field before applying current formatting.

MLM_QUERYFORMATTEXTLENGTH
Return the number of text characters, in the entry field after applying current formatting.

MLM_SETIMPORTEXPORT
Set the address and size of buffer for import and export text.

MLM_QUERYIMPORTEXPORT
Returns the values from the most recent MLM_SETIMPORTEXPORT or 0 if they have never been set.

MLM_IMPORT
Insert import buffer text into entry field.

MLM_EXPORT
Copy text from the entry field to the export buffer.

MLM_DELETE
Delete a number of characters from the entry field.

MLM_SETSEL
Set the zero-based offsets of the start and end of the selected text.

MLM_QUERYSEL
Return the zero-based offsets of the start and end of selected text.

MLM_UNDO
Undo the last operation.

MLM_QUERYUNDO
Return whether undo is possible.

MLM_RESETUNDO
Set status of undo so that last operation cannot be undone.

MLM_SETFONT
Set display to use given font.

MLM_QUERYFONT
Return current font structure.

MLM_SETTEXTCOLOR
Set color for entry field text.

MLM_QUERYTEXTCOLOR
Return color for entry field text.

MLM_SETBACKCOLOR
Set background color for entry field.

MLM_QUERYBACKCOLOR
Return background color of entry field.

MLM_QUERYFIRSTCHAR
Return zero-based offset of first visible character.

MLM_SETFIRSTCHAR
Scroll to make requested character first visible character.

MLM_COPY
Copy selected text from the field to the clipboard, in MLE_CFTEXT format.

MLM_CUT
Send selected text from the field to the clipboard, and delete it from the field.

MLM_PASTE
Copy selected text from the clipboard into the field.

MLM_CLEAR
Delete selected text.

MLM_ENABLEREFRESH
Enable screen refreshes.

MLM_DISABLEREFRESH
Disable screen refreshes.

MLM_SEARCH
Search for specified text string.

Scrolling Increment for Arrow Keys in Entry Fields

For arrow keys, this is one character at a time. When keying text, this is a third of the field width (rounded up to allow for 2-character fields).

Combination Boxes

A combination box (also known as *combo box*) is a single control that combines the function of an entry field with a list box. This control is effective when information must be entered through an entry field but a list of possible choices is available.

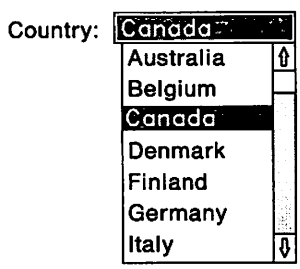


Figure 7-6. Combination Box

Combination boxes are WC_COMBOBOX class windows and can have the following CBS_combobox styles:

CBS_SIMPLE
Designates a list box/entry field pair such that both controls are always visible. When the selection changes in the list box, it is placed in the entry field. Also, the text in the entry field is completed with the closest match from the list box.

CBS_DROPDOWN
Inherits all the CBS_SIMPLE properties. In addition, the list box is hidden until the user clicks on the prompt box to display the list box.

CBS_DROPDOWNLIST
A variation in which the entry field is replaced by static text that serves to display the current selection from the list box. The list box must be displayed by the user.

There are three types of combination box:

- Simple Combination Box
- Drop-down Combination Box
- Drop-down List.

Simple Combination Boxes

A simple combination box is one in which the list box is permanently displayed beneath and slightly to the right of the entry field.

Drop-down Combination Boxes

A drop-down combination box is a variation of the simple combination box in which the list box is hidden until the user elects to display it. A prompt box containing a downward pointing arrow is positioned to the right of the entry field. This shows the user that the control is a drop-down combination box rather than a simple entry field, and that a list is hidden.



Figure 7-7. Drop-down Combination Box

Drop-down Lists

A drop-down list is similar to a drop-down combination box except that it has a single selection field in place of an entry field.

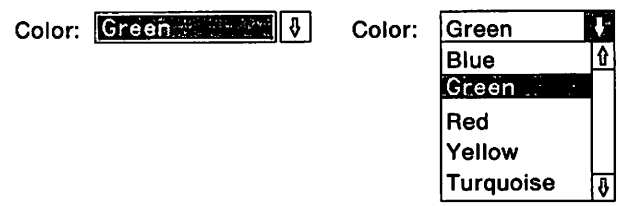


Figure 7-8. Drop-down List

Receiving Combination Box Notification Messages

The combination box sends a WM_CONTROL notification message to its owner window whenever a user interacts with it.

Combination Box Notification Codes

CBN_EFCHANGE
Contents of entry field have changed

CBN_EFSCROLL
Contents of entry field have scrolled

CBN_MEMERROR
Memory allocation error has occurred

CBN_LBSELECT
An item in the list box has been selected

CBN_LBSCROLL
The list box has scrolled

CBN_SHOWLIST
The list box is being shown (dropped down)

CBN_ENTER
The user selects ENTER or double clicks on a list box item.

Sending and Posting Combination Box Control Messages

The combination box control supports the same messages that are supported by entry field and list box controls. The following additional messages are also supported:

CBM_SHOWLIST Used to show or hide the list box field of a combination box. This message has no effect on simple combination boxes as the list box is automatically shown.

CBM_ISLISTSHOWING Sent to determine whether the list box field of a combination box is showing.

CBM_HILITE Highlights or unhighlights the prompt of a combination box.

LM_SETCURSOR Sets the cursor to a specified item in the list box. It does not alter the current selection.

LM_QUERYCURSOR Returns the index of the item where the cursor is set. If no item has the cursor, it returns LIT_NONE.

LM_SETSELECTION Sets the selection of a specified item.

WM_UPDATESTYLE Updates the style of a window. It sets the style bits specified by afMaks to values specified by afData and redraws the window to reflect the updated style.

Scrolling Panel Contents — Scroll Bars

If the contents of a window can be scrolled, you should supply scroll bars. You can also use scroll bars as a slider control where the user needs to select a position within a range. For example, the control panel of the user shell displays scroll bars so the user can adjust colors.

When all the available data is displayed, there is no need to show a scroll bar.

When the data is already scrolled as far as possible in one direction the scroll arrows in that direction should be grayed.

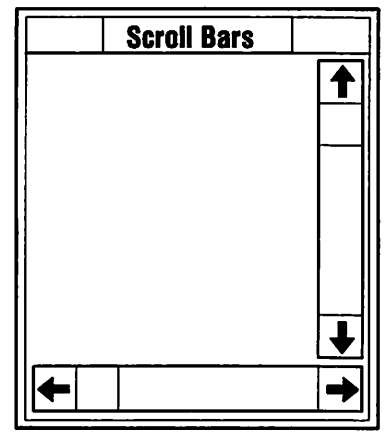


Figure 7-9. Scroll Bars in a Window

A scroll bar is a WC_SCROLLBAR class window, that can be either horizontally or vertically oriented, as defined by the SBS_HORZ or SBS_VERT scroll bar style. A scroll bar can be positioned anywhere in a dialog box or window, and is usually owned by the dialog box or window. A window containing horizontal and vertical scroll bars is shown in Figure 7-9.

OS/2 itself uses scroll bars as optional elements of a standard window, and in list boxes. Whenever the slider position changes, WM_VSCROLL or

WM_HSCROLL notification messages are sent to the owner window.

Control and Notification Messages: These are the same as those listed in "Scroll Bars" on page 5-16.

Displaying Errors — Message Boxes

Message boxes are a form of dialog box used to display warnings, errors, or information to the user.

A message box consists of a window with static text and one of several arrangements of pushbuttons. The text consists of a simple message to the user. The pushbuttons enable the user to choose one of a simple set of responses to the message. The buttons displayed are specified by your application from a standard set, and include such actions as **OK** to continue normally, and **Cancel** to halt an operation without completing it. All the pushbuttons can be selected by a mnemonic, except Cancel, which is selected by the Esc key, and Enter or OK, which are both selected by the Enter key.

Message boxes typically are displayed only for a short period during some particular interaction. When the interaction is completed, the message box is removed from the screen.

Message boxes are simple in user interface terms. They are equally simple in programming terms, because you do not need to define them in a dialog template. Instead, the WinMessageBox (WCMBOX) call specifies the contents and appearance. For example, the code:

```
Response = WinMessageBox(HWND_DESKTOP, hwndFrame,
           "Select OK to finish the job.", NULL,
           1, MB_OKCANCEL|MB_HELP);
```

produces the result in Figure 7-10.

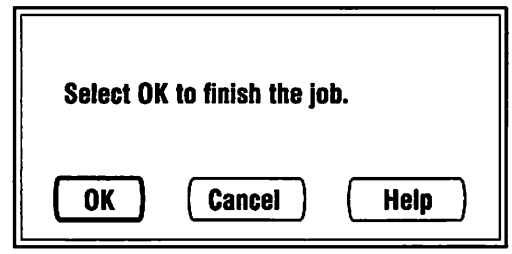


Figure 7-10. Example of a Message Box

However, you do not usually code the text of the message into the call, as in this example. Instead, you can put text strings in a resource file, and load them as needed using the WinLoadString (WMLSTR) call. You can use the WinSubstituteStrings (WUSUBS) call to replace variables in a text string with other text. The use of text string resources is described in "Text-String Tables" on page 8-4.

A message box initially appears in the center of the screen. The only variations possible with the message box are the text displayed, and which of the standard formats is used. The format is controlled by the MB_ values used as the last parameter on the call. Common User Access defines three types of message and makes recommendations for the contents of the message boxes and their pushbuttons. The three types are information, warning, and error messages. For warning or error messages, you can use the call WinAlarm (WMALRM) to provide an audible beep for the user.

Message boxes are displayed with a background color of SYSCLR_WINDOW. If you want to display messages with color, use dialog boxes.

Message Box Pushbuttons

The following MB_ values specify the pushbuttons that appear:

MB_OK OK

MB_ENTER Enter

MB_CANCEL Cancel

MB_HELP Help

MB_OKCANCEL OK and Cancel

MB_ENTERCANCEL Enter and Cancel

MB_RETRYCANCEL Retry and Cancel

MB_YESNO Yes and No

MB_YESNOCANCEL
Yes, No, and Cancel

MB_ABORTRETRYIGNORE
Abort, Retry, and Ignore.

The following MB_ values specify the default selection:

MB_DEFBUTTON1 First button

MB_DEFBUTTON2 Second button

MB_DEFBUTTON3 Third button.

If no button is specified as the default selection, the system assumes MB_DEFBUTTON1.

The response value returned by WinMessageBox (WCMBOX) shows which pushbutton the user selected. For example, the response is MBID_ENTER if **Enter** is selected.

MBID_OK Ok

MBID_ENTER Enter

MBID_CANCEL Cancel

MBID_RETRY Retry

MBID_YES Yes

MBID_NO No

MBID_ABORT Abort

MBID_RETRY Retry

MBID_IGNORE Ignore.

The Help pushbutton is a special case. When the user selects Help, a WM_HELP message is sent to the window procedure of the message box.

Message Box Icons

The following MB_ values specify the icon that appears:

MB_INFORMATION
Black information 'i' in a square box

MB_QUERY Question mark in a square box

MB_WARNING Black '!' in a square box

MB_ERROR 'STOP' sign on white background

MB_CRITICAL 'STOP' sign on white background.

The following message box types are still supported but should not be used in new applications:

MB_NOICON
MB_ICONHAND
MB_ICONASTERISK
MB_ICONQUESTION
MB_ICONEXCLAMATION.

Movable Message Boxes

MB_MOVABLE Movable.

A movable message box is displayed with a title bar and system menu. It can be moved with either the mouse or keyboard. The title of the message appears in the title bar. The user can get to the system menu by pressing the Shift+Esc keys. Only the Move, Close, and Task List choices are shown, with their accelerators. When the user selects Close on the system menu (or double clicks on it) the message box is removed, and MBID_CANCEL is returned to the application whether or not a Cancel pushbutton is present.

Message Box Modality

The following MB_ values specify the modality of a message box:

MB_APPLMODAL Application modal

MB_SYSTEMMODAL System modal.

Because a message box is a type of dialog box, it is subject to the some of the modality considerations discussed in "Modeless and Modal Dialogs" on page 7-2. However, a message box should *always* be modal. A message box can be system-modal, as specified by the MB_SYSTEMMODAL indicator, or application modal, as specified by MB_APPLMODAL, which is the default. As for dialog boxes, you can use the parent and owner parameters on the WinMessageBox (WCMBOX) call to control the extent to which the user is prevented from interacting with other screen objects. To conform with the Common User Access, critical messages should appear in a pop-up window. This can be implemented by an application-modal message box with the desktop as its parent, and the frame window as its owner.

Static Controls

Static controls are non-interactive. They are used simply to enhance the appearance of a dialog box, and to provide information to the user in the form of text strings. There are six static controls:

1. Group Boxes
2. Text
3. Frames
4. Rectangles
5. Icons
6. Bit maps.

A **Group Box** is a rectangular frame with a title which is used to group related choices (they should not be used to group pushbuttons or list boxes). A group box provides a visual aid to users by isolating and naming groups of controls that work together. Multiple sets of choices or a single set of choices may be contained within a group box.

Text displays a character string in the dialog box, which can be used for giving instructions or displaying messages.

Frames are solid rectangular frames that can be used to create simple graphics in a dialog box.

Rectangles are filled rectangles that can be used as a background for other static elements.

Icon creates a rectangle that marks the position for an icon, while **Bit map** creates a rectangle that marks the position for a bit map.

The Dialog Box Editor

You can use the *Dialog Box Editor*, supplied in the OS/2 1.2 Programming Tools, to create and modify dialog boxes. You can also use it to create and modify the controls and text within a dialog box. The Dialog Box Editor is a *what-you-see-is-what-you-get* (WYSIWYG) editor, used to create the dialog interactively. You need a mouse to use it. As you create the dialog box and its controls you see them on the screen as the user sees them when your program is run. You can place each dialog box and its controls where you want them on the screen. In addition, you can test the dialog box in a simple way before you incorporate it into your application.

Each dialog box and control can have either an integer identifier or a symbolic identifier that equates to an integer identifier, either of which you can use in your application to refer to the dialog boxes or controls. If you intend using symbolic identifiers in your application, you must enter the symbolic and integer identifiers in an include file. If you do not use symbolic names, the Dialog Box Editor supplies an integer identifier for each control, and for the dialog box itself. You can use the Dialog Box Editor to create the include file, or you can use a text editor to create the include file before using the Dialog Box Editor.

It is good programming practice to plan the resources that your application will use, and choose a naming and numbering convention for the symbolic or integer identifiers *before* you create them. You should keep the include file separate from other include files used by your application. Include files used by the Dialog Box Editor can contain only #define statements that define their symbolic identifiers and equivalent integers.

Although the Dialog Box Editor draws dialog boxes and controls on the screen so you can see what they look like when used by your application, it does *not* save them as graphics. Instead, the Dialog Box Editor saves them in an ASCII-text format file, that has a .DLG extension. This is the dialog template, mentioned at the beginning of the chapter. The Dialog Box Editor also creates a compiled form of the .DLG file, in a file with a .RES extension. The .DLG and .RES files can contain more than one dialog box. The resource file may contain other application resources, such as icons, bit maps, and string tables. It is attached to the application's executable (.EXE) file during the compile and link process.

Dialog Box Layout

Dialog boxes should be designed to identify clearly the information that the user is required to complete. The following are a few Common User Access guidelines:

- Lay out the controls in columns, starting at the upper-left corner, for left to right or top to bottom scanning.
- Vertically and horizontally align selection and entry fields so that the cursor moves in a straight line.
- Arrange the controls in the sequence that the user would complete them.

- If there are only a few entry fields, locate them at the top of the dialog box.
- Make groups of controls obvious by use of group boxes and white space.
- Align group boxes, where possible. Group boxes can be extended to the right to line up with other group boxes.
- Use field identifiers to identify the purpose of single and multiple groups of choices.

For further information regarding dialog boxes, refer to the Common User Access publications.

Creating a Dialog Box

To run the Dialog Box Editor, display **Desktop Manager**, select the **Toolkit Editors** group, and select **Dialog Box Editor**. A copyright information panel is displayed. Select **Enter**, and a window is displayed that contains the choices **File**, **Include**, **Edit**, **Control**, **Options**, and **Exit**. On-line help that tells you how to use the editor can be accessed from most of the windows.

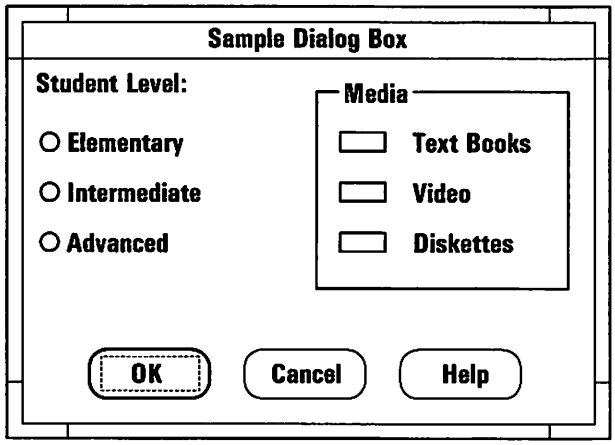


Figure 7-11. Sample Dialog Box

To create a dialog box similar to the one in Figure 7-11 follow these steps:

1. You may either:
 - Select **New** from the **Include** pull-down. This opens a new include file, but the Dialog Box Editor does not tell you that it has opened the include file. Then select **New Dialog...** from the **Edit** pull-down. The editor opens new files with the extensions .RES and .DLG.

 or you may:
 - Select **New** from the **File** pull-down. This opens new resource files with the extensions .RES and .DLG, but the Dialog Box Editor does not tell you that it has opened the resource files. You can open a new include file or an existing one. Select **Yes** to open a new include file. Select **New Dialog...** from the **Edit** pull-down. Both courses of action have the same effect.

 When you edit a dialog box, the names of the resource and include files are shown in the title bar of the Dialog Box Editor. If you are editing a new file that has not yet been named or saved, **Untitled** appears in the title bar in place of a name. An asterisk (*) by the side of the file name means that there are unsaved changes.
2. The **Dialog Box ID** pop-up appears. A default integer number is supplied in the entry field. Type in a symbolic identifier for the dialog box, (MYDIALOG, for example) then a space, then the integer number. Click on **Enter** to place them both in the include file.
3. A dialog box appears in the lower-left corner of the editor screen, as shown in Figure 7-12. This dialog box is what you edit.

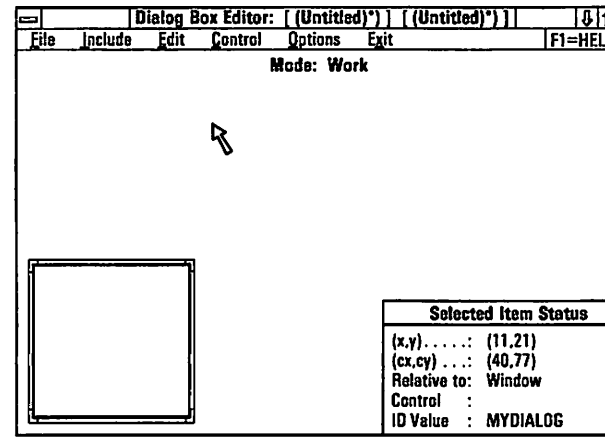


Figure 7-12. The Dialog Box Editor

The dialog box is enclosed by a frame. This indicates that the dialog box is selected for editing. Whenever a new dialog box is created, this selection is automatic, but at other times you must click on the dialog box or **Control** to select it. The editor supplies the frame so you can change the width and height of the dialog box. This frame does not appear in the final version of the dialog box. Make the dialog box larger by clicking on its borders and dragging until the box is about five inches wide and four

inches high. This can be done in one operation by clicking on the upper-right corner of the frame and dragging diagonally upwards and to the right.

You can reposition the entire dialog box by moving the pointer inside the area enclosed by the frame, holding the mouse button down, and dragging the shadow box across the screen. When the shadow box is in the position where you want the dialog box to appear, release the mouse button. The dialog box appears in that position. Alternatively, you can move the dialog box using the keyboard arrow keys. You can reposition the dialog box at any time during the edit.

Information about the item you are editing is displayed in the **Selected Item Status** box in the lower-right corner of the editor window. As you move the shadow box, the x,y coordinates change. These are the coordinates of the origin of the dialog box relative to the origin of the window. The cx,cy coordinates are the width and height of the dialog box. The symbolic identifier is also shown.

4. Select **Styles...** from the **Edit** pull-down. The **Dialog Box Styles** pop-up window appears. The **Dialog Frame** check box is already checked. Leave it this way, so you get a frame on the final dialog box that the user sees.
5. Select Title Bar, then click on the **Dialog Title** field, and type the text **Sample dialog box** into the **Dialog Title** entry field. Click on **Enter**. The title appears at the top of your dialog box.
6. You now have your dialog box, and are ready to put in controls.

Setting a Grid — Grid Option

Controls are placed in a dialog box using the mouse, and can be moved in line with each other also using the mouse. However, controls can be positioned more accurately using the keyboard arrow keys or mouse after the Grid values have been set. **Grid** lets you set the number of character spaces (in dialog units) by which you can move dialog boxes and controls when using the dialog box editor.

1. Select **Grid** from the **Options** pull-down. The **Grid Values** pop-up is displayed as shown in Figure 7-13. The initial settings are one unit for x and one for y.
2. Change the x setting to 10 and the y setting to 5. Click on **Enter**.

The horizontal (x) and vertical (y) values are in dialog units. A horizontal dialog unit is 0.25 of the standard character size. A vertical dialog unit is 0.125 of the standard character size. For example, if you move a control to the left or the right (using the mouse or keyboard arrow keys) with x set at 20, it moves in steps of twenty dialog units.

When you subsequently position dialog boxes or controls, the objects move by the specified number of dialog units on an invisible grid. Large values make it easier to align controls with each other, while small values allow you to position controls in the dialog box more precisely.

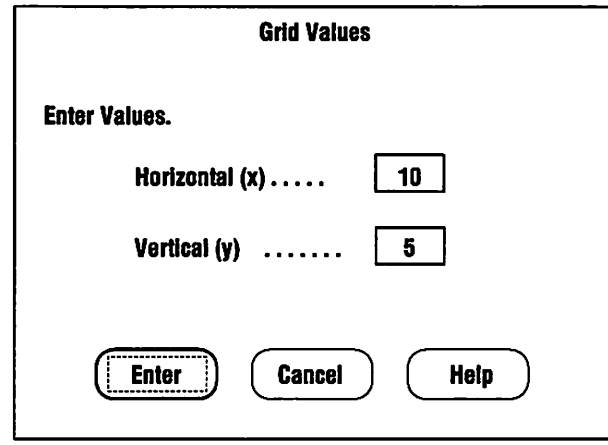


Figure 7-13. Grid Values

Entering Controls

The control pull-down lists all the controls that you can put in a dialog box. Dynamic controls precede static controls in this list.

1. Select **Text**. The pointer becomes a small cross in a square. The center of the cross marks the position where the lower-left corner of the control's frame will be set. Position the cross inside the dialog box near the upper-left corner and click the mouse. The **Static Styles** pop-up is displayed. Type **Student Level:** in the **Text** entry field. Observe that the next sequential integer is supplied in the **ID symbol** entry field. Click on **Enter**.
2. Replace the integer with **ID_NULL** and click on **Enter**. A message box asks:

```
This symbol does not exist.  Use
the default ID?
```

This means that the editor has looked in the include file and has not found a symbolic identifier with the name ID_NULL. Click on **OK**. The editor assigns the default identifier to the symbolic identifier you entered, and places it in the include file. This is another technique for entering symbolic identifiers.

3. If you want to view or change the include file at any time, select **View Include...** from the **Include** pull-down. The **View Include File** pop-up appears. The symbolic and integer identifier for the dialog box and the text control are displayed in the list box. The pop-up allows you to add, delete, and change the identifiers, and view the hexadecimal equivalents of the integers. Select the **Enter** pushbutton to remove the pop-up and register any changes. Select **Escape** if you have not made any changes.

4. In your dialog box, the size of the static control is not large enough for you to see all the text. To remedy this, click on the text, and a frame appears around it. Drag the right-hand edge of the frame to the right to enlarge the field. When you release the mouse button you should be able to see all the text. When a control has a frame around it, it is selected and you can use a shadow box to position it, as you did with the dialog box.

5. To add another control, select **Radio Button** from the **Control** pull-down and position the cross cursor just beneath the **Student Level** text. Click on **Enter**.

 The **Button Control Styles** pop-up is displayed. Type **Elementary** in the **Button Text** entry field and click on **Enter**. Drag the right edge of the frame that surrounds the radio button until you can see all of the text.

 Select another **Radio Button** and type **Intermediate** in the **Button Text** entry field. Position this radio button below the first one.

 Select another radio button and type **Advanced** in the **Button Text** entry field. Position this radio button below the other two.

6. Select **Group Box** from the **Control** pull-down. Position the cross cursor to the right of the column of radio buttons (approximately halfway across the dialog box) and click on **Enter**. The **Static Styles** pop-up is displayed. Type **Media** in the **Text** entry field and click on **Enter**. This is the title of the group box. Click on the lower-right corner of the group box frame and drag it diagonally down and to the right to enlarge it. The bottom of the group box frame should be lower than the last of the radio buttons, and the right-hand side of the group box should be almost at the far right of the dialog box. This is to make room for a group of check boxes that will go inside the group box. When you use group boxes to group controls, you always create the group box before the controls that are to go inside it.

7. Select **Check Box** from the **Control** pull-down. Position the cursor inside the group box in line with the first radio button in the list and click the mouse. The **Button Control Styles** pop-up is displayed. Type **Text Books** in the **Button Text** entry field and click on **Enter**. Enlarge the frame of the check box until all of the text is displayed.

 Select **Check Box** again and position the cross cursor below the first check box. Type **Video** in the **Button Text** entry field and click **Enter**. Enlarge the check box frame as before.

 To add a third **Check Box**, position the cross cursor below the previous two check boxes and type **Diskettes** in the **Button Text** entry field.

 In the left-hand side of the dialog box you should now have a column of radio buttons with a heading of **Student Level**, and on the right a group box with a heading of **Media** and containing three check boxes.

8. Finally, add three pushbuttons to the dialog box. Select **Pushbutton** from the **Control** pull-down. Position the cross cursor in the lower-left side of the dialog box and click the mouse. Type **OK** in the **Button Text** entry field of the **Button Control Styles** pop-up and click on **Enter**.

 Position another pushbutton to the right of the first one (in the lower middle of the dialog box) and type **Cancel** in the **Button Text** entry field.

 Select a third pushbutton and position it to the right of the second. Type **Help** in the **Button Text** entry field.

The dialog box and its controls are now complete.

Try selecting each of the controls, and observe the information in the **Selected Item Status**. It holds information about each control that you edit.

Testing the Dialog Box

To test the dialog box, select **Test Mode** from the **Options** pull-down. The dialog box is displayed as it appears to the user from a program. In test mode, you can select controls, and their appearance changes in the same way as they do in an application. To return to work mode, click on **Test Mode** again to de-select it.

Group Control Ordering – Order Groups

This option allows you to gather controls into groups and to change the order in which the tab keys and arrow keys move the selection cursor around the controls. It is good practice to put group markers around all separate groups of controls, including putting a marker before the first control in the list.

The list box shows the order in which the selection cursor moves between the controls when the user presses the arrow and tab keys. (The coordinate position of a control when displayed in the dialog box does not affect the order.) Initially, the controls are listed in the order in which they were created.

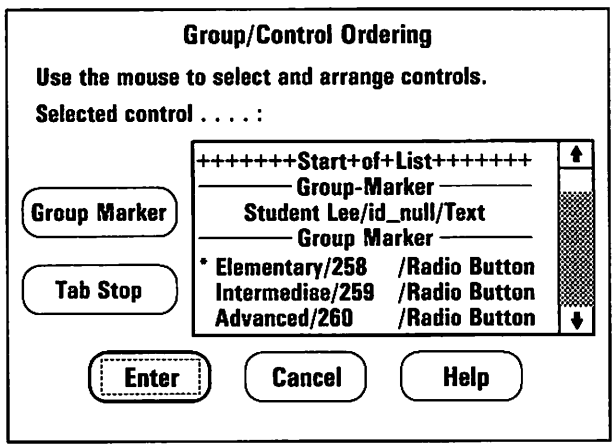


Figure 7-14. Group Control Ordering.

1. Select **Order Groups** from the **Options** pull-down. The **Group Control Ordering** pop-up is displayed as illustrated in Figure 7-14. Click the pointer on the first radio button in the list box. Click on the **Group Marker** pushbutton. A group marker is now displayed between the **Text** control and the first radio button in the list.

 Scroll down the list and click on the first pushbutton in the list. Click on the **Group Marker** pushbutton. This has organized your controls into groups of text, radio buttons, check boxes, and pushbuttons.

2. Now set the tab-stops. The controls marked with an asterisk already have tab-stops. You will need to delete the tab-stops from the second and third radio button and check box.

 Click on the second radio button in the list to mark it, and then click on the **Delete Tab** pushbutton. Repeat this for the third radio button. Perform the same operation for the second and third check box in the list. When this is complete, click on **Enter**.

The editing of the dialog box is now complete and you can test it.

Try moving the controls in the list to see how the changes affect the movement of the cursor during testing. To change the position of a control in the list, click on its name to select it. Then position the pointer in the list where you want the name to appear. The pointer changes shape to a short horizontal line when it is over a place where you can insert the name. To insert the control name, click the mouse button.

Edit an Existing Dialog Box

To go back and change the dialog box or controls, move the pointer into your dialog box, and click on the item to select it for editing. A frame appears around the item. Note that clicking inside the dialog box and outside a control selects the dialog box. To edit the item, you can select the items in the **Edit** pull-down. For example, you can select **Style** to change the style of the selected control or dialog box. You can also undo what you have created, using **Clear Control** or **Clear Dialog**.

To create another dialog box in the same resource file, select **New Dialog** from the **Edit** pull-down. Your existing dialog box is saved.

Ending an Edit Session

To end the edit session, select **Exit Dialog Box Editor** from the **Exit** pull-down. You see prompts for the file names of the files you want to save. The next time you use the editor, if you want to edit the same file select **Open...** from the **Files** pull-down.

Example Code: Dialog Template File for MYDIALOG

```
DLGINCLUDE 1 "DBE.H"

DLGTEMPLATE mydialog LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Sample Dialog Box", mydialog, 11, 8, 170, 105, FS_NOBYTEALIGN |
            FS_DLGBORDER | WS_VISIBLE | WS_SAVEBITS, FCF_TITLEBAR
    BEGIN
        CONTROL "Student Level:", id_null, -1, 94, 63, 9, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
        CONTROL "Elementary", 258, 7, 82, 62, 11, WC_BUTTON, BS_RADIOBUTTON |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Intermediate", 259, 7, 67, 73, 9, WC_BUTTON, BS_RADIOBUTTON |
                WS_VISIBLE
        CONTROL "Advanced", 260, 7, 51, 52, 13, WC_BUTTON, BS_RADIOBUTTON |
                WS_VISIBLE
        CONTROL "Media", 261, 87, 48, 75, 54, WC_STATIC, SS_GROUPBOX |
                WS_GROUP | WS_VISIBLE
        CONTROL "Text books", 262, 97, 83, 60, 10, WC_BUTTON, BS_CHECKBOX |
                WS_TABSTOP | WS_VISIBLE
        CONTROL "Video", 263, 97, 68, 46, 10, WC_BUTTON, BS_CHECKBOX |
                WS_VISIBLE
        CONTROL "CBT", 264, 97, 53, 32, 10, WC_BUTTON, BS_CHECKBOX | WS_VISIBLE
        CONTROL "OK", 265, 7, 20, 38, 12, WC_BUTTON, BS_PUSHBUTTON | WS_GROUP |
                WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", 266, 61, 20, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_TABSTOP | WS_VISIBLE
        CONTROL "Help", 267, 117, 20, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_TABSTOP | WS_VISIBLE
    END
END
```

The Dialog Template

The Dialog Box Editor creates an ASCII text file that has a file-name extension of .DLG. It also creates a compiled form of the .DLG file: this has a file-name extension of .RES.

The .DLG file contains a series of statements, collectively termed a *dialog template*, that define each dialog box and each control in each dialog box. The statement for each dialog box contains the data required to create it, namely its class, size, position, window text, and any other special information required for the window.

Normally, the template consists of a dialog box window followed by the controls contained within it, as child windows.

The DLGINCLUDE statement specifies the file name of the include file.

The DLGTEMPLATE statement specifies the symbolic identifier of the dialog box (MYDIALOG), and loading and memory options. The actual dialog template is contained by the first BEGIN and last END statement. There is a CONTROL statement for each of the controls in the dialog box. The CONTROL statement is a general statement that is followed by parameters that further specify the control:

- Text, where appropriate. For example, the text **OK** is defined for one of the pushbuttons.
- Application-defined symbolic or integer identifiers for each control. Your application uses the identifier to track the responses from con-

trols. For example, ID_NULL is the identifier of the text control.

- The types and positions of the various controls. For example, the group box control is a control window of window class WC_STATIC. The **Cancel** and **Help** pushbuttons are of window class WC_BUTTON.
- The appearance and operation of the dialog box and its controls are specified in detail by combinations of style parameters. For example, the check boxes have a class style of BS_CHECKBOX, and radio buttons have a class style of BS_RADIOBUTTON. You can also specify appropriate WS_ styles.

If necessary, you can use a text editor to edit the .DLG file, for example, to *fine-tune* the dialog template produced by the dialog box editor. You can even use a text editor to produce your own .DLG file. The syntax of all the statements used to specify dialog boxes and controls are listed in the *Programming Reference: Volume 2*. The dialog box editor always uses the general CONTROL statement with window classes and control styles to define controls. You can use the CONTROL statement in the same way to define your controls, or you can use any of several predefined control statements that give you the same result. For example, the predefined control statement PUSHBUTTON gives you a WC_BUTTON class window with default styles of BS_PUSHBUTTON and WS_TABSTOP.

A dialog template can be in either:

- A resource file, (generated from the .DLG or .RES file) in which case the calls you use depend on the modality issue discussed in "Modeless and Modal Dialogs" on page 7-2,

or

- A block of memory that has the DLGTEMPLATE data structure, in which case you use WinCreateDlg (WCCRD) to create the dialog box from the template.

The dialog template uses device-independent *dialog units* for the coordinate system that defines the layout of controls in the dialog box. A dialog unit is expressed in terms of the *default standard character size*, which can vary from device to device. You do not need to put code in your application to reformat the dialog box when displaying it on different devices. (Dialogs may need editing if a different system font is loaded.) A horizontal dialog unit is 0.25 of the standard character size. A vertical dialog unit is 0.125 of the standard character size. Dialog units are expressed as offsets from the origin of the dialog box, which is the lower-left corner.

A dialog template is a general structure. It equally could be termed a window template, because you can use it to define any window in an application. If you prefer, use the statement WINDOWTEMPLATE instead of DLGTEMPLATE, because it is identical functionally. This could reduce the initialization phase of the application to registering the application window classes and calling WinLoadDlg (WCLD) to load the template.

If you use the dialog box editor to define a standard window, you will have to edit the resulting .DLG file, to ensure that you have a client window and the required parent-child relationships. You will also have to use WinLoadMenu (WCLM) in your application, to create an action bar for the window, because you cannot create menus using the Dialog Box Editor.

The .RES file is an object-format compiled version of the .DLG file, created when the Dialog Box Editor calls a compiler called the *Resource Compiler*. The Dialog Box Editor uses the .RES file as input on any subsequent edit of the same dialog. This means that, if you use a text editor to fine-tune a .DLG file, and you want subsequently to reedit the dialog using the Dialog Box Editor, you must first use the Resource Compiler to generate a new .RES file from the .DLG file.

Your application can use either the .RES file output by the Dialog Box Editor, or a .RES file created from the .DLG file and the other resources. See "Creating a Resource File and Binding It to Your .EXE File" on page 8-6 for details. If your application uses the .DLG file, it must be included by your application's resource script file.

The rcinclude statement includes the .DLG file created by the Dialog Box Editor; for example:

```
rcinclude dbe.dlg /* Includes .dlg file */
```

The corresponding .H file created by the Dialog Box Editor must also be included in the .RC file.

To see how the rcinclude statement fits in with other resource definitions, and how to include the .H file, see "Writing a Resource Script File" on page 8-5.

Using OS/2-defined control windows, OS/2 draws and operates for your application the controls specified in the resource file. Controls are windows and can be used within any other window. (This is described in "Using Controls Without Dialog Boxes" and in "Creating Your Own Controls.")

Chapter 9, "DIALOG1 Dialog Box Sample Application" describes how the sample application uses a dialog box containing an entry field and a pair of pushbuttons. Chapter 10, "The DIALOG2 Sample Application" describes how the sample uses several dialog boxes that use most of the standard controls.

Using Controls Without Dialog Boxes

When you want to use a control like a radio button in a window, you do not have to go through the entire process of specifying the dialog box and the radio button in a resource file. Instead, you can use the WinCreateWindow (WICRT) call to create the control.

Specify the window class for the control you require (for example, WC_BUTTON for a button control) and the control style (for example, BS_AUTORADIOBUTTON). Such class names are preregistered in OS/2. The way you use these classes is language-dependent:

- In C/2 and Macro Assembler/2 you can specify a class using a WC_ constant. You can also use the form *#nnnnn*, where *nnnnn* is one through five digits corresponding to the value of the class name shown in the header files. For example, a button control (WC_BUTTON) is specified by #3.
- In COBOL/2 and FORTRAN/2 the class names must be specified as string constants; for example, `WC-BUTTON = '#3'`. You can copy these constants from the relevant constants file.

If you are creating the control as a child of another window, you should specify that window as the parent and the owner. The owner window requires a dialog procedure to process the notification messages from the control. There are examples of dialog procedures in Chapter 9, "DIALOG1 Dialog Box Sample Application" and Chapter 10, "The DIALOG2 Sample Application." The equivalent COBOL/2 dialog routines are described in Chapter 11, "The DLG1 Nonreentrant Sample Application" and Chapter 12, "The DLG2 Nonreentrant Sample Application."

When you create multiple controls using WinCreateWindow (WICRT), you should also use a window style of WS_CLIPSIBLINGS for the controls. This ensures correct repainting of the controls if they overlap.

Creating Your Own Controls

In addition to the standard controls provided by the OS/2 API, an application can provide its own controls.

User controls can take two forms. In the first, the application defines its own extensions to OS/2-defined controls. In the second, the application defines its own control window classes.

Extensions to OS/2-defined control types generally mean the application drawing objects other than the default text within the control. Examples of this are where the application needs to display a bit map as a menu choice, or an item in a list box. Such items are known as *ownerdraw* items, because the owner window typically has to arrange for the drawing of the item. When you specify MIS_OWNERDRAW for menus, or LS_OWNERDRAW for list boxes, a WM_MEASUREITEM message is sent to the owner when the owner is created, so that it can calculate the height of the item. A WM_DRAWITEM message is sent to the owner to tell it to draw the item.

Application-defined control window classes can be as general as an application requires, because they are windows like any others. However, they can be used in many different places within a single application and can have a range of control and notification messages that are unique to their class.

Chapter 8. Resources and Resource Files

Chapter 5, "The Standard Window," describes how to define menus and accelerator tables as resources in a resource script file. Chapter 7, "Dialog Boxes and Controls" describes how to create dialog box resources, using the Dialog Box Editor or a text editor, and how to include them in the resource script file. This chapter describes the other resources that you can create using the Programming Tools:

- Icons, pointers, and bit maps
- Fonts.

It describes how to refer from your program to the resource script file that contains these resources and your dialog boxes. It also tells you how to define the following resources in the resource script file:

- Text-string tables
- Help tables.

The Icon Editor

The Icon Editor lets you create your own figures (icons, pointers, and bit maps), and save them for use by applications.

Icons, pointers, and bit maps produced by the icon editor are graphic symbols comprised of pels in any of the following display states:

- Black
- White
- Color
- Screen (background color)
- Inverse screen (inverse of background color).

The size of bit maps created by the Icon Editor can be in the range 1 × 1 through 99 × 99 pels. You define the size of a bit map when you create it.

An application can use an icon to represent a minimized standard window. For example, an application that lists telephone numbers could use a ☎ icon when minimized. An application can also use icons as warning symbols in message boxes, for example, an exclamation mark or an upraised hand.

An application can associate a pointer with the mouse, or a similar pointing device, so that the user can move the pointer around the screen, and select controls or text. A pointer could also be used to draw graphics on the screen, in an interactive graphics application. For example, a free-hand graphics drawing application could use a pencil shape to represent the pointer.

To run the Icon Editor, display **Desktop Manager**, select the **Toolkit Editors** group, and select **Icon Editor**. A copyright information panel is displayed. Select **Enter**, and the following panel is displayed:

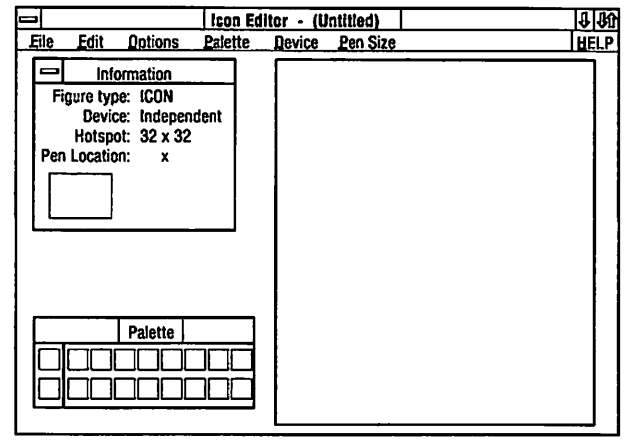


Figure 8-1. The Icon Editor

The information window (in the upper left-hand corner of the icon editor window) shows details of the current figure that you are editing, and a true-scale image of that figure.

The palette window (lower left-hand corner) displays the colors that are available for use during editing, and shows which color is currently selected (the selected color has a frame around it).

The editing window on the right is your working area. Use the mouse or keyboard to move the pointer, clicking or dragging it to paint the enlarged representation of pels with the selected color.

Creating a Figure

To create a new icon, pointer, or bit map, select **New** from the **File** pull-down. The **New Figure** pop-up appears, prompting you for further information, as illustrated in Figure 8-2.

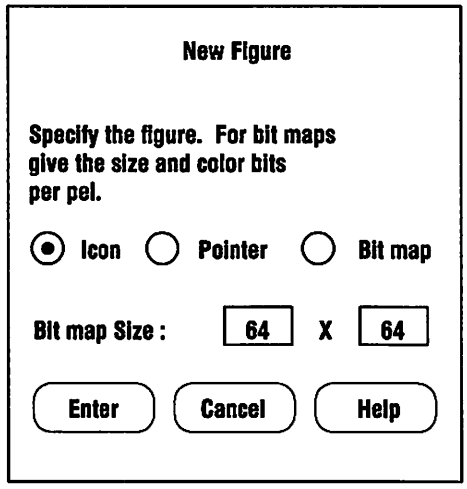


Figure 8-2. New Figure

Select the figure type: Icon, Pointer, or Bit map. For a bit map you must specify the width and height in pels (1 × 1 through 99 × 99). Select **Enter**.

There is a facility called **Grid** in the **Options** pull-down that allows you to superimpose a grid over the editing window. This can be useful when you want to draw a symmetrical figure. Each cell of the grid represents one pel in the figure.

There is also a facility available that can be used for straight line drawing. It is **Draw Straight** in the **Options** pull-down. When it has been invoked, it allows the pointer to draw or erase horizontal and vertical lines only. Even if you deviate from the horizontal row, a horizontal line is produced when the mouse pointer is dragged across the editing window. Dragging the mouse up or down produces straight vertical lines.

The **Pensize** menu allows you to specify how many pels the pointer paints at a time for icons, pointers, and bit maps. You may choose from four available pen sizes:

- Very Small (1 x 1)
- Small (2 x 2)
- Medium (3 x 3)
- Large (5 x 5)
- Extra Large (7 x 7).

The **Palette** window allows you to specify the pen, and background colors for the figure. There are sixteen default pen colors displayed in the largest section of the palette window. The two colors shown beneath the system menu icon in the palette window show the background color and inverse color.

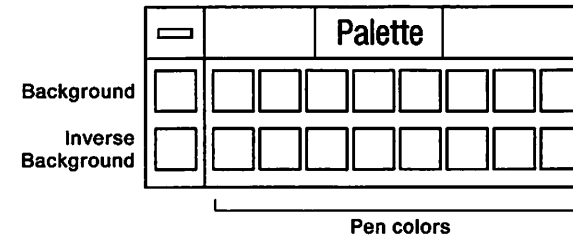


Figure 8-3. Color Palette

You can alter the choice of colors by editing the colors in the palette. Select the color to be edited with the mouse; a frame appears around the selected color. Select **Edit** from the **Palette** pull-down. The **Color Selector** pop-up appears and displays three scroll bars with three associated edit controls that allow you to re-define the value associated with the currently selected color.

Alternatively, double-clicking on any color in the palette causes the color selector pop-up to appear without using the palette pull-down.

You may specify the color in two formats: as a mixture of primary colors, or in the Hue, Saturation, and Value format. You may use an exact RGB value in the edit controls, or use the scroll bars to select the color mix.

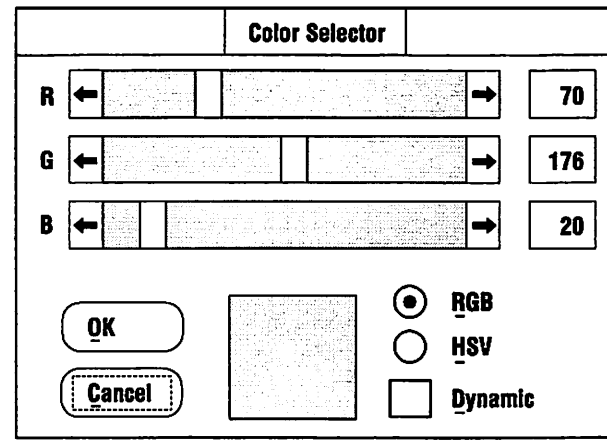


Figure 8-4. Color Selector

For an icon or pointer, the Icon Editor specifies that one of the pels is the **hot spot**. An application uses the hot spot to determine the position of an icon on the screen, or to define a pointer's current screen coordinates. When an application uses WinQueryPointerPos (WMQPTP) to query the screen position of a pointer, OS/2 returns the coordinates of the pointer's hot spot. The default location for the hotspot is in the center of the icon or pointer. The choice is only available when editing icons or pointers; bit maps do not have hotspots. When you select hotspot from the **Edit** pull-down, the cursor changes shape, and the screen coordinates of the current hotspot are displayed in the information window.

When you edit a pointer, **Test** causes the mouse pointer to change to the currently displayed pointer that you are editing. If you are editing an icon, Test causes the Icon Editor to display the icon that you are editing as its icon.

The **Hide Info** option toggles the visibility of the information window.

The **X bkgnd** option causes all screen or inverse screen colors to be overdrawn with a *x*. This can be used to distinguish transparent pels from color pels. This option does not apply to bit maps because they have no transparent pels.

Multiple Device Support

The Icon Editor edits and saves a device-independent form of the icon. However, the **Device** facility enables you to create versions of the icon for specific display devices. The Device pull-down displays a choice of three functions: create a new device form, select an existing form, and delete a form.

An independent form is automatically created when you create a new icon or pointer and all other forms are derived from it. If you select any of the other device forms listed in the pull-down, a new form is created for the specified device. The **Custom** option enables you to create an icon or pointer for any other device.

The **List** option lists all existing forms, including custom and standard forms. Any of the items in this list may be selected and edited. The **Delete** option deletes the selected device form. However, the device-independent form cannot be deleted.

Several icon bit maps may be saved in a single icon resource, and when the icon is saved, all versions are saved with it in a format that includes a device resolution tag for each version. When the icon is loaded from a resource file, the display device resolution is matched against the device for which each device-dependent icon was intended. If a match is found, that icon is used. If no match is found, the application uses the device-independent icon, which always exists.

Figure files may contain any of the following forms to support multiple devices:

- Independent
- CGA (2 colors)
- EGA (16 colors)
- VGA (16 colors)
- 8514 (256 colors)
- Custom.

Device-dependent icons are icons that are designed for a particular display resolution.

An application can display icons or bit maps in dialog boxes.

The WinLoadPointer (WMLPOI) loads a pointer, icon, or bit map created by the Icon Editor. WinCreatePointer (WICPOI) creates a pointer from the loaded pointer.

Depending on the resource you are creating, the icon editor produces a file with either the extension .ICO for icons, .PTR for pointers, or .BMP for bit maps. The .ICO, .PTR, or .BMP files must be referred to in the resource script file for your application. The external files containing icons, pointers, and bit maps are all referenced in the resource script file by single-line statements that have a similar format; for example:

```
ICON    ID_MAINWND myprog.ico   /* Icon      */
POINTER ID_PTR     mypoint.ptr  /* Pointer   */
BITMAP  ID_BMP     mybtmp.bmp   /* Bit map   */
```

ICON, POINTER, and BITMAP keywords identify the resource type.

The resource type is followed by a symbolic name or integer identifier that is used by your application to identify the resource. For example, with ICON, the ID_MAINWND identifier can be used by the application in the control data parameter of the WinCreateWindow (WICRT) call (or as a parameter to the WinCreateStdWindow (WICRTS) call) that creates the frame of your application's main window. OS/2 then associates the icon with the main window.

The symbolic name or identifier can be followed by loading and memory options. The options are not used by the example, as it lets the options default.

The last part of the statement is the file name and file type of the resource created by the Icon Editor. A fully qualified path name must be given if the file is not in the current directory. An icon that it used for a minimized application main window should have the same file name as the application's executable file. To see how the icon, pointer, and bit-map definitions fit with other resource definitions, see "Writing a Resource Script File" on page 8-5.

Associating an Icon with a Text-Windowed or Full-Screen Application

When a new text-windowed or full screen session is started, and the user has not explicitly supplied an icon file, the system looks for an .ICO file in the same subdirectory as the executable program, and uses that icon file instead of the default OS/2 or VIO icon.

If CMD.EXE is to be run in the new session, the program name (for the purposes of looking for the icon file) will be the string that follows /K (or /C) in the argument string. Thus, for

```
START "title" /C myfile
```

the icon file used is MYFILE.ICO.

If an error occurs while loading the icon file, the icon file is ignored and the default icon used. Note that, if an error occurs when loading an icon explicitly given on the DosStartSession API call, the call returns an error.

Text-String Tables

You use a text editor to define text strings in the resource script file. Here is the string table from the resource script file used by HELLO2:

```
STRINGTABLE PRELOAD
BEGIN
 IDS_HELLO, "Hello"
 IDS_1,     "--1--"
 IDS_2,     "--2--"
 IDS_3,     "--3--"
END
```

You can use variables within a text string. Mark the positions of variables in the string using substrings of the form "%n," where n is an integer in the range 0-9.

The statements for specifying strings are listed in the *Programming Reference: Volume 2*. To see how the string definitions fit in with other resource definitions, see "Writing a Resource Script File" on page 8-5.

Note: The maximum number of characters allowed in a resource-file string (including the NULL termination character) is 256. The substrings that mark the positions of variables in a string count as two characters each. If the text in your resource file is to be translated into other national languages, remember that the other languages may need longer strings.

When the compiled form of the resource script file has been linked to the application, use the WinLoadString (WMLSTR) call to bring a specific string into your program. If you want to substitute a substring for a variable after you have loaded a string, use the WinSubstituteStrings (WUSUBS) call. Parameters to this call are the memory locations of the initial and final strings. The WinSubstituteStrings call sends a WM_SUBSTITUTESTRING message. Your application must provide the text of the substitution string so that the message can return it to the WinSubstituteStrings call.

Help Panels and Tables

Help panels are defined using information tag language, and converted into resource files by the information tag language compiler. See "Part 3. Presentation Manager Applications – Using OS/2 Information Presentation Facility" for details.

Help tables tell the Help Manager which help panels to display for which windows and fields in windows. Help tables do not define the contents of the help panels. See "Part 3. Presentation Manager Applications – Using OS/2 Information Presentation Facility" for details.

Writing a Resource Script File

You use a text editor to create a resource script file that has a .RC extension. The .RC file is the input file to the Resource Compiler – a compiler that converts all the resources into an executable form, and places them in a file that has a .RES extension. A .RC file can contain statements that define resources, and statements that include resources from other files. Usually, you create text-based resources such as menus, accelerators, and text strings in the .RC file itself, but specify only the file names of external files such as the dialog box .DLG file, and files that contain non-text-based resources such as icons, pointers, and bit maps. You treat fonts differently, as described in "The Font Editor" on page 8-8.

You must make sure that none of the include files in your resource script file contain an End of File marker. When the Resource Compiler sees an End of File Marker it assumes it to be the end of all input.

Example Code: Skeleton Resource Script File

Here is an outline of a .RC file, leaving out a lot of the detailed statements, to give you an impression of what a fully specified file looks like:

```
#include <os2.h>                    /* Directive  */
#include "mybox.h"                  /* Directive  */

ICON    ID_MAINWND myprog.ico       /* Icon       */

POINTER ID_PTR     mypoint.ptr      /* Pointer    */

BITMAP  ID_BMP     mybtmp.bmp       /* Bit map    */

rcinclude mybox.dlg                 /* Dialog box */

MENU ID_MENU PRELOAD                /* Menu       */
BEGIN
  .
  .
  .
  .
END

ACCELTABLE ID_ACCEL PRELOAD         /*Accelerators*/
BEGIN
  .
  .
```

```
 .
 .
END

STRINGTABLE                          /* Strings    */
BEGIN
 .
 .
 .
 .
END

HELPTABLE  TABLE_1
BEGIN
 .
 .
 .
 .
END

HELPSUBTABLE  SUBTABLE_1
BEGIN
 .
 .
 .
 .
END
```

All the statements that appear in a resource script file are listed in the *Programming Reference: Volume 2*.

The above example contains directive statements. The directives define actions for the Resource Compiler to complete before it compiles the resources. (They are based on the preprocessor directives used with the C/2 compiler.) You can use the directives to assign values to names, include the contents of files, and control compilation of the resources. The example .RC file includes the OS2.H file, that contains the symbolic constants used by OS/2 (so it must be included) and MYBOX.H, which represents the .H file created by the Dialog Box Editor, and containing the symbolic constants used in the .RC file. A resource directive can be #include, #define, #undef, #ifdef, #ifndef, #if, #elif, #else, #endif, dlginclude, rcdata, and rcinclude.

Creating a Resource File and Binding It to Your .EXE File

This section describes two alternative methods for creating a resource file and binding it to the .EXE file that contains your application.

Method 1 – Putting All the Resources in the .RC File

Follow these steps, illustrated in Figure 8-5 on page 8-7, to create and bind resources to your application:

1. Use the Programming Tools to create the icons, bit maps, pointers, dialog boxes, and controls that your application requires.
2. Use a text editor to create a .RC file that refers to the files containing all the above resources, and defines the menus, accelerators, and text strings.
3. Run the two stages of the Resource Compiler together, using the .RC file and the .EXE file as input. The Resource Compiler compiles the resource file in the first stage, and binds the compiled resources to your application .EXE file in the second stage. The result is a .EXE file containing your program and its resources.

Creating a .EXE file, and the options you specify to run the Resource Compiler, are fully described in the *Building Programs* manual.

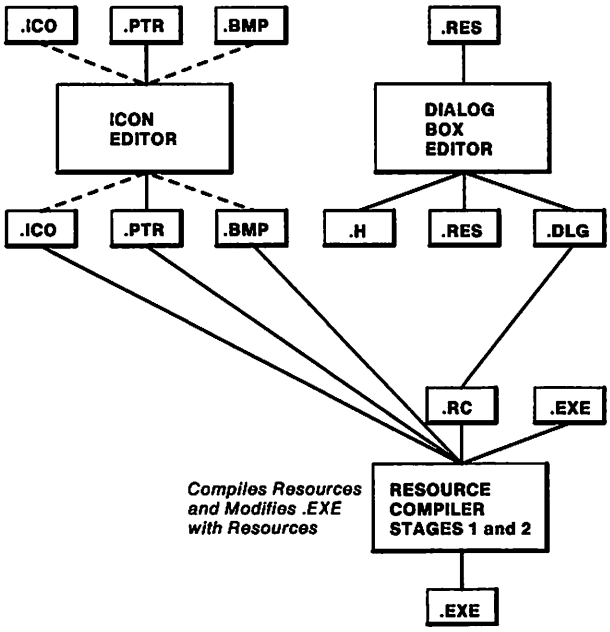


Figure 8-5. Using the Resource Editors and Resource Compiler – Method 1

Method 2 – Putting All the Resources Except Dialog Boxes in the .RC File

In this method, follow these steps, illustrated in Figure 8-6, to create a .RES file and use it to modify your application:

1. Use the Programming Tools to create the icons, bit maps, and pointers, that your application requires. (You are going to create the dialog boxes and controls later.)
2. Use a text editor to create a .RC file that refers to the files containing all the above resources, and defines the menus, accelerators, and text strings.
3. Run the first stage of the Resource Compiler, using the .RC file as input, to create a .RES file.
4. Use the Dialog Box Editor, with the .RES file from the previous step as input. (Remember that the Dialog Box Editor uses a .RES file as input.) Create the dialog boxes and controls that your application requires. The resulting file is a .RES file containing compiled versions of your resources.
5. Run the second stage of the Resource Compiler, using the .RES file from the Dialog Box Editor and the .EXE file as input, to create a .EXE file containing your program and its resources.

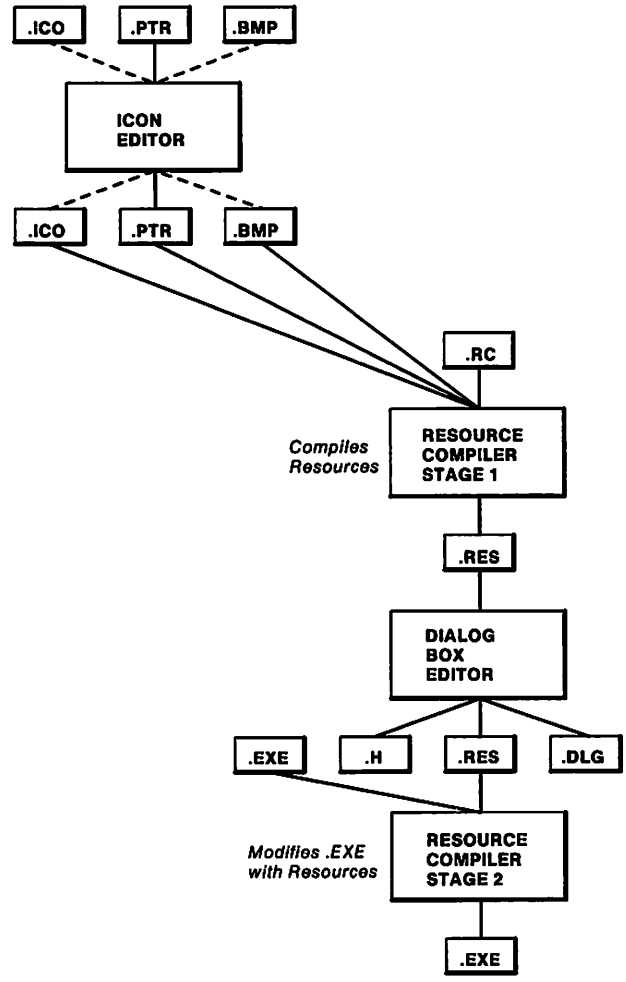


Figure 8-6. Using the Resource Editors and Resource Compiler – Method 2

Comparing Method 1 with Method 2

Method 1 involves the least steps if you choose to run the two stages of the Resource Compiler together. Method 2 is the best method if you later want to change the dialog boxes, because you can use the previous .RES file as input to the Dialog Box Editor, and rerun the second stage of the Resource Compiler.

The Font Editor

A font is a set of alphanumeric characters, punctuation characters, and other symbols that share a common typeface design and line weight. An application loads a font from a dynamic link library file (.DLL file). This section describes how to produce such a file.

You can use the Font Editor in the OS/2 1.2 Programming Tools to design and save your own fonts for use in applications. You edit an enlarged version of the letter, character, or symbol in an editing window, using the mouse to switch the enlarged representation of pels to black or white. You can change a series of pels by dragging the mouse pointer through them while holding down the mouse button. A true-scale version is shown in a viewing window to the right of the edit window.

To run the Font Editor, display **Desktop Manager**, select the **Toolkit Editors** group, and select **Font Editor**. A copyright information panel is displayed. Select **Enter**, and the following panel appears:

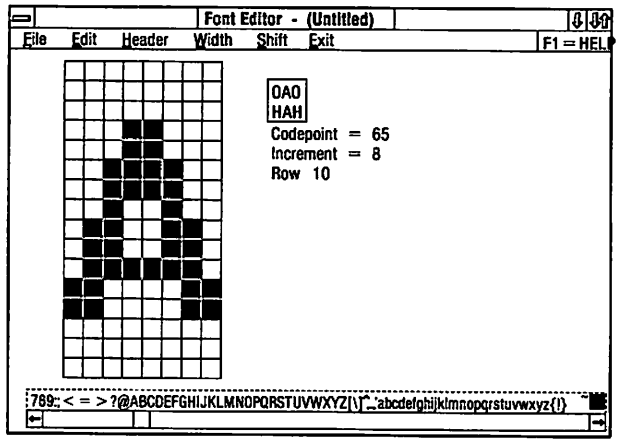


Figure 8-7. The Font Editor

To edit a font, select one of the options in the **File** menu to open a new or existing font. The letter "A" appears in both the editing and viewing windows. The rest of the font appears in the character selection scroll box at the bottom of the font editor window.

To edit any other character in the font, select it from the character selection scroll box. The character appears in the editing and viewing windows.

Editing Functions

There are several functions available that enable you to tailor individual fonts. These functions are options in the **Header**, **Width**, and **Shift** pull-downs.

Type Definition

Use the options in the **Header** pull-down to define the type that you want to create. The **General** option lets you specify spacing (fixed or proportional), type face style, line width, and type weight.

Use the **Naming** option to specify the identification details such as the type-face name.

You may specify the font character dimensions in the **Sizes** option, and the position of characters in the **Relations** option.

The **Definitions** option lets you change the character spacing of a character in a proportional font.

Character Width

The options in the **Width** pull-down let you alter the width of a single character. This menu is enabled only when the font that you are editing is a proportional space font. You can make a character wider or narrower by adding or deleting columns of pels from the right, the left, or both sides. You may also use the **Set Increment** option to set the width of a character. The font editor contains on-line help panels that tell you how to perform these functions.

Shift

The options in the **Shift** pull-down enable you to insert or delete a one-pel-wide row or column in the character that you are editing. When you select shift, the pointer becomes a flat horizontal or vertical bar when inside the edit window. This enables you to position it exactly where you want the operation to take place.

Also available is the **Cancel Choice** option which enables you to cancel a shift mode without adding or deleting pels.

Font Resource File

The font editor creates a file with a .FNT extension. The .FNT file is not referred to in the same resource file as other resources. Instead, it has its own resource file that contains a single-line statement that has a similar format to the ICON, POINTER, and BITMAP statements, for example:

```
FONT    101         myfont.fnt      /* Font     */
```

The FONT keyword identifies the resource type.

The resource type is followed by an integer identifier that is used by the application to identify the resource. The integer is used as a parameter to the WinCreateStdWindow (WICRTS) call. You cannot use a symbolic name for a font.

The integer identifier can be followed by loading and memory options. Again, the example lets them default.

The last part of the statement is the file name of the resource created by the font editor. A full path name must be given if it is not in the current directory.

"Creating a Resource File and Binding It to Your .EXE File" on page 8-6 told you how to bind resources to an .EXE file. The process of producing a font file is similar, as shown in Figure 8-8. You bind one or more .FNT files to a dummy .DLL, to produce a file containing the font or fonts. The final file should have the extension .FON.

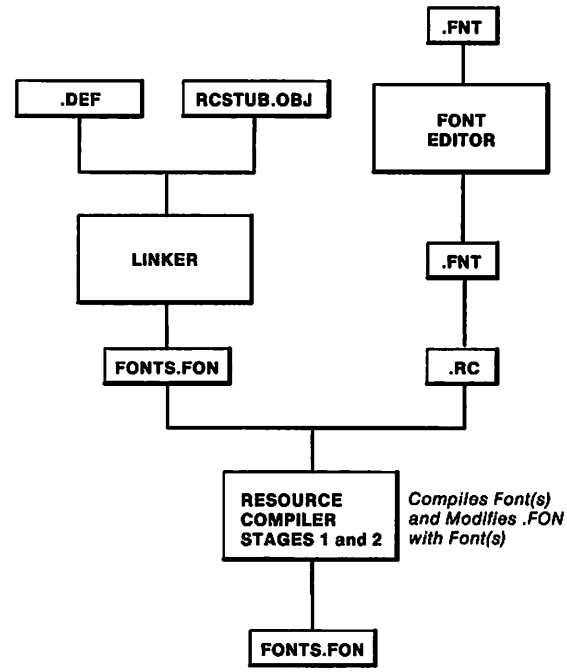


Figure 8-8. Using the Font Editor, Linker, and Resource Compiler to Produce a .FON File

The process of putting resources into a dynamic link library is described in the *Building Programs* manual. The .FON file created by the process can be installed on the system. This can be done either by using the Control Panel in the User Shell, or by an installing application. In both cases it becomes a *public* font that can be used by any application in the system.

If a font is not installed on the system, it is a *private* font. Before your application can use the font, your application must use GpiLoadFonts to load the .FON file. For more information about using graphics text in an application, see Chapter 23, "Graphics Text."

Putting a Resource File in a Dynamic Link Library Module

Instead of binding a resource file to your application, you can place your resources in a .DLL, that has to be loaded at run time. The process is similar to that for putting one or more fonts in a .DLL. The process of putting resources into a dynamic link library is described in the *Building Programs* manual.

The process involves creating, assembling, and linking a .DLL file that you use instead of the .EXE file as input to the Resource Compiler, for method 1 or 2. The .DLL file can be empty or, where you are creating a library of window procedures, can contain the procedures and the resources they use. Either way, you can load the resulting .DLL file into your application using DosLoadModule or WinLoadLibrary. These calls return a handle, that you use as a parameter to the following calls to load their respective resources:

GpiLoadBitmap
Bit maps

WinLoadAcceltable (WMLACT)
Accelerator tables

WinDlgBox (WCLMD) or WinLoadDlg (WCLD)
Dialog boxes

WinLoadMenu (WCLM)
Menus

WinLoadPointer (WMLPOI)
Pointers, Icons, and Bit Maps created by the Icon Editor

WinLoadString (WMLSTR)
Strings.

There are advantages in putting your resource file in a dynamic-link library so that it is separate from your application. For example, you can have several national language versions of resource files, but you need only one application. If you do not put the national language resource files in .DLL files in a dynamic-link library, you need a version of your application for every language you support.

Chapter 9. DIALOG1 Dialog Box Sample Application

This chapter describes the DIALOG1 sample application, a simple application that uses a dialog box to obtain data from the user. The program uses a standard window to display its main window. The standard window has an action bar with the menu item **Options**. The client area has a blank white background.

When the user selects the **Options** item in the action bar, a pull-down appears, containing the items **Where...** and **Exit**, as shown in Figure 9-1.

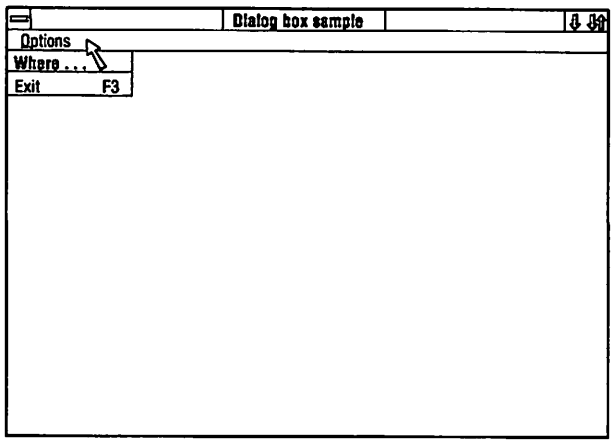


Figure 9-1. DIALOG1 Sample Application Window

The ellipsis (...) indicates to the user that a dialog box follows from the item. When the user selects **Where...**, the application responds by displaying a dialog box.

The dialog box contains the text **Where do you live?**, and a text entry field that is blank. The dialog box also contains two pushbuttons, one with the text **OK**, the other with the text **Cancel**. Selecting **Cancel** always causes the dialog box to disappear without any other effect.

Text can be entered into the entry field, for example, **Winchester** as shown in Figure 9-2.

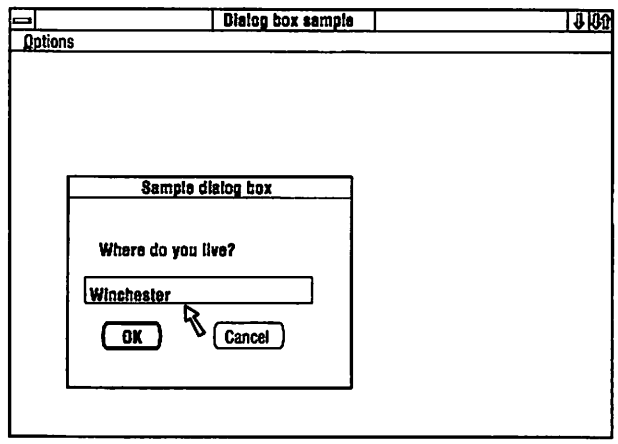


Figure 9-2. DIALOG1 Sample Application Window with Dialog Box

When the user selects the **OK** pushbutton, the dialog box disappears and the name of the town or city appears in the window, at the end of a string of text; for example:

You live in
Winchester

The **Exit** option on the pull-down menu causes the program to terminate.

Resource Script File

The resource file in DIALOG1 contains the definitions of the following resources:

- Icon, used to represent the main window when it is minimized
- Accelerator table
- Action bar and pull-down
- Text string.

Example Code: DIALOG1 Resource Script File

```
#include <os2.h>
#include "dialog1.h"

ICON       ID_MAINWND PRELOAD dialog1.ico

ACCELTABLE ID_MAINWND PRELOAD
BEGIN
  VK_F3, ID_EXITPROGRAM, VIRTUALKEY
END

MENU       ID_MAINWND PRELOAD
BEGIN
  SUBMENU "~Options",             ID_COMMANDS
  BEGIN
    MENUITEM "~Where...",         ID_WHERE,       MIS_TEXT
    MENUITEM SEPARATOR
    MENUITEM "E~xit \tF3",        ID_EXITPROGRAM, MIS_TEXT
  END
END

STRINGTABLE PRELOAD
BEGIN
  ID_RESPONSE, "You live in "
END

rcinclude dialog1.dlg
```

Dialog Template

Example Code: DIALOG1 Dialog Template File

In the resource file, the rcinclude statement points to the file DIALOG1.DLG created by the dialog box editor.

```
DLGINCLUDE 1 "DIALOG1.H"

DLGTEMPLATE ID_MYDIALOG PRELOAD MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Sample dialog box", ID_MYDIALOG, 91, 90, 111, 59,
           WS_VISIBLE | WS_SAVEBITS,
           FCF_DLGBORDER | FCF_NOBYTEALIGN | FCF_TITLEBAR | FCF_SYSMENU
    BEGIN
        CONTROL "Where do you live?", ID_NULL, 9, 43, 91, 10, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "", ID_ENTRYFIELD, 10, 27, 87, 8, WC_ENTRYFIELD,
                ES_LEFT | ES_AUTOSCROLL | ES_MARGIN |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "OK", 1, 6, 4, 40, 12, WC_BUTTON,
                BS_PUSHBUTTON | BS_DEFAULT | WS_GROUP | WS_TABSTOP |
                WS_VISIBLE
        CONTROL "Cancel", 2, 60, 4, 40, 12, WC_BUTTON,
                BS_PUSHBUTTON | WS_VISIBLE
    END
END
```

Dialog Include File

Example Code: Dialog Include File

Here is the .H file that equates the symbolic identifiers used to identify the resources to their integer values:

```
#define ID_MAINWND      256
#define ID_MYDIALOG     257
#define ID_ENTRYFIELD   258
#define ID_COMMANDS     259
#define ID_WHERE        260
#define ID_EXIT         261
#define ID_EXITPROGRAM  262
#define ID_RESUME       263
#define ID_RESPONSE     264
#define ID_NULL         9999
```

Example Code: DIALOG1 Sample Application

```
#define INCL_WINSYS                     /* Selectively include          */
#define INCL_WINFRAMEMGR                /* relevant parts of            */
#define INCL_WINMENUS                   /* the PM header file           */
#define INCL_GPI
#include <os2.h>
#include <string.h>                     /* C/2 string functions         */
#include "dialog1.h"                    /* Resource symbolic identifiers*/

#define LENGTH_STRING  33               /* Length of string             */

/* Function prototypes                                                  */
MRESULT EXPENTRY MyWindowProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 );
MRESULT EXPENTRY MyDlgProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 );
VOID cdecl main( VOID );
VOID SetSystemMenu( HWND hwnd );
                                        /* Define parameters by type    */
HAB   hab;                              /* Anchor block handle          */
HMQ   hmq;                              /* Message queue handle         */
HWND  hwndClient;                       /* Client window handle         */
ULONG flCreate = FCF_STANDARD;          /* Frame control flag           */
BOOL  bComplete = FALSE;                /* Switch for first time through*/
HWND  hwndFrame;                        /* Frame window handle          */
CHAR  szLocation[LENGTH_STRING] = "";   /* To hold text entered by user.*/
CHAR  szResponse[LENGTH_STRING] = "";   /* To hold "You live in".       */

/**********************  Start of main procedure  ***********************/
VOID cdecl main(  )
{
  QMSG  qmsg;

  hab = WinInitialize( NULL );
  hmq = WinCreateMsgQueue( hab, 0 );

  WinRegisterClass(                     /* Register Window Class        */
      hab,                              /* Anchor block handle          */
      "MyWindow",                       /* Window Class name            */
      MyWindowProc,                     /* Address of Window Procedure  */
      CS_SIZEREDRAW,                    /* Class Style                  */
      0                                 /* No extra window words        */
      );

  hwndFrame = WinCreateStdWindow(
        HWND_DESKTOP,                   /* Desktop Window is parent     */
        WS_VISIBLE,                     /* Frame Style                  */
        &flCreate,                      /* Frame Control Flag           */
        "MyWindow",                     /* Window Class name            */
        "",                             /* No window title              */
        0L,                             /* Client style of visible      */
        NULL,                           /* Resource is in .EXE file     */
        ID_MAINWND,                     /* Frame window identifier      */
        &hwndClient                     /* Client window handle         */
        );

/**********************************************************************/
/* Get and dispatch messages from the application message queue       */
```

```
/* until WinGetMsg returns FALSE, indicating a WM_QUIT message.        */
/*********************************************************************/
  while( WinGetMsg( hab, &qmsg, (HWND)NULL, 0, 0 ) )
    WinDispatchMsg( hab, &qmsg );

  WinDestroyWindow( hwndFrame );        /* Tidy up...                  */
  WinDestroyMsgQueue( hmq );            /* and                         */
  WinTerminate( hab );                  /* terminate the application   */
}
/*********************  End of main procedure  ************************/

/*******************  Start of window procedure  *********************/
MRESULT EXPENTRY MyWindowProc ( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
  HPS    hps;                           /* Presentation space handle   */
  POINTL pt;                            /* String screen coordinates   */
  RECTL  rc;                            /* Window rectangle            */
  switch( msg )
  {
    case WM_CREATE:
      /* Window initialization - load "You live in" string from resource*/
      WinLoadString( hab,
                     (HMODULE)NULL,        /* Resource is in .EXE file  */
                     ID_RESPONSE,          /* String identifier         */
                     sizeof( szResponse ), /* Size of buffer            */
                     (PSZ)szResponse       /* Buffer                    */
                   );
      break;
    case WM_COMMAND:
      /*****************************************************************/
      /* For the Commands pull-down, when the user chooses "Where...",  */
      /* the application displays a dialog box. WinDlgBox calls the     */
      /* dialog procedure below. WinDlgBox does not return until the    */
      /* dialog procedure returns.                                      */
      /* WinInvalidateRegion sends a WM_PAINT message.                  */
      /* For the Exit pull-down, the application posts itself a WM_CLOSE*/
      /* message if Exit is chosen.                                     */
      /*****************************************************************/
      switch ( SHORT1FROMMP( mp1 ) )
      {
        case ID_WHERE:
          WinDlgBox( HWND_DESKTOP,      /* Place anywhere on desktop   */
                     hwndFrame,         /* Owned by frame              */
                     MyDlgProc,         /* Address of dialog procedure */
                     NULL,              /* Module handle               */
                     ID_MYDIALOG,       /* Dialog identifier in resource*/
                     NULL );            /* Initialization data         */
          WinInvalidateRegion( hwnd, NULL, FALSE ); /* Force a repaint */
          break;
        case ID_EXITPROGRAM:
          WinPostMsg( hwnd, WM_CLOSE, 0L, 0L );
          break;
        default:
      return WinDefWindowProc( hwnd, msg, mp1, mp2 );
      }
      break;
    case WM_PAINT:
      /*****************************************************************/
      /* Window contents are drawn here. First time through, bComplete  */
      /* is FALSE, so window is simply filled with SYSCLR_WINDOW.       */
```

```
      /* On subsequent passes, if bComplete has been set to TRUE in the */
      /* dialog procedure, GpiCharStringAt draws the text.              */
      /*****************************************************************/
      hps = WinBeginPaint( hwnd, (HPS)NULL, &rc );
      WinFillRect( hps, &rc, SYSCLR_WINDOW );
      if( bComplete)
      {
        GpiSetColor( hps, CLR_NEUTRAL ); /* Text color same as PM       */
        pt.x = 50L; pt.y = 50L;          /* Coordinates of "You live in" */
        GpiCharStringAt( hps, &pt, (LONG)strlen( szResponse ), szResponse );
        pt.x = 50L; pt.y = 30L;          /* Coordinates of location      */
        GpiCharStringAt( hps, &pt, (LONG)strlen( szLocation ), szLocation );
      }
      WinEndPaint( hps );                /* Drawing is complete          */
      break;
    case WM_CLOSE:
      WinPostMsg( hwnd, WM_QUIT, 0L, 0L );      /* Cause termination    */
      break;
    default:
      /* Must have WinDefWindowProc in your window procedure            */
      return WinDefWindowProc( hwnd, msg, mp1, mp2 );
  }
  return FALSE;
}
/********************** End of window procedure  **********************/

/********************** Start of dialog procedure  ********************/
MRESULT EXPENTRY MyDlgProc( HWND hwndDlg, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
  switch ( msg )
  {
    case WM_INITDLG:
      /*****************************************************************/
      /* PM sends a WM_INITDLG message when the dialog box is created   */
      /*****************************************************************/
      SetSystemMenu( hwndDlg );          /* Edit entries in system menu  */
      break;

    case WM_COMMAND:                     /* Posted by pushbutton or key  */
      /*****************************************************************/
      /* PM sends a WM_COMMAND message when the user presses either     */
      /* the Enter or Escape pushbuttons.                               */
      /*****************************************************************/
      switch( SHORT1FROMMP( mp1 ) )      /* Extract the command value    */
      {
        case DID_OK:                     /* The Enter pushbutton or key. */
                                         /* WinQueryWindowText copies the*/
                                         /* contents of the entry field  */
                                         /* into szLocation.             */
          WinQueryWindowText( WinWindowFromID( hwndDlg, ID_ENTRYFIELD ),
                              LENGTH_STRING,
                              szLocation );
          bComplete = TRUE;              /* Set switch so strings are    */
                                         /* drawn in WM_PAINT processing.*/
        case DID_CANCEL:          /* The Cancel pushbutton or Escape key */
          WinDismissDlg( hwndDlg, TRUE );  /* Removes the dialog box    */
          return FALSE;
        default:
          break;
      }
```

```
      break;
    default:
      /*******************************************************************/
      /* Any event messages that the dialog procedure has not processed */
      /* come here and are processed by WinDefDlgProc.                   */
      /* This call MUST exist in your dialog procedure.                  */
      /*******************************************************************/
      return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
  }
  return FALSE;
}
/********************** End of dialog procedure  ************************/
/******************* Start of SetSystemMenu procedure  ******************/
VOID SetSystemMenu( HWND hwndDlg )
{
  HWND     hwndSysMenu;                /* system menu pulldown handle  */
  MENUITEM miTemp;                     /* menu item template           */
  SHORT    sItemId;                    /* system menu item ID          */
  SHORT    sItemIndex;                 /* system menu item index       */

  /*********************************************************************/
  /* Get the handle of the system menu pulldown.                       */
  /*********************************************************************/
  hwndSysMenu = WinWindowFromID( hwndDlg, FID_SYSMENU );
  WinSendMsg( hwndSysMenu,
              MM_QUERYITEM,
              MPFROM2SHORT( SC_SYSMENU, FALSE ),
              MPFROMP( (PSZ)&miTemp ));
  hwndSysMenu = miTemp.hwndSubMenu;

  /*********************************************************************/
  /* Remove all items from the system menu pull-down that we don't want. */
  /*********************************************************************/
  for ( sItemIndex = 0, sItemId = 0; sItemId != MIT_ERROR; sItemIndex++)
  {
    sItemId = (SHORT)WinSendMsg( hwndSysMenu,
                                 MM_ITEMIDFROMPOSITION,
                                 MPFROMSHORT( sItemIndex ),
                                (MPARAM)NULL );
    if ( sItemId != MIT_ERROR &&
         sItemId != SC_MOVE   &&
         sItemId != SC_CLOSE  &&
         sItemId != SC_TASKMANAGER )
    {
      WinSendMsg( hwndSysMenu,
                  MM_DELETEITEM,
                  MPFROM2SHORT( sItemId, FALSE ),
                 (MPARAM)NULL );
      sItemIndex--;
    }
  }

}
/******************* End of SetSystemMenu procedure  ********************/
```

DIALOG1 Window Procedure MyWindowProc

DIALOG1's window procedure explicitly processes only the WM_CREATE, WM_COMMAND, WM_PAINT, and WM_CLOSE messages, and passes all other messages to the default window procedure.

The WM_CREATE Message

PM sends the WM_CREATE message to the window when the program creates it. The application processes WM_CREATE by loading the string ID_RESPONSE from the resource file.

The WM_COMMAND Message

The command processing handles the pull-down choices. When the user selects the **Where...** choice from the **Commands** pull-down, a WM_COMMAND message with a command value of ID_WHERE is posted to the application. When the user selects the **Exit Dialog1** choice from the **Exit** pull-down, a WM_COMMAND message with a command value of ID_EXITPROGRAM is posted to the application.

The ID_WHERE Command Value

The **Where...** command value ID_WHERE is processed by calling WinDlgBox to load and display the dialog box. WinDlgBox (WCLMD) specifies the desktop as the parent of the dialog box, and the frame as the owner of the dialog box. This makes the dialog modal, as described in "Modeless and Modal Dialogs" on page 7-2. User interaction with the frame and all its descendants is prevented. This conforms with the Common User Access definition for a pop-up window. The control does not return from WinDlgBox until the processing of the dialog box is complete. WinDlgBox supplies the address of the application dialog procedure as a parameter.

A dialog procedure has the same structure as a window procedure. The main difference is that instead of the application calling WinDefWindowProc (WMDWP) for messages not handled by the application dialog procedure, it calls WinDefDlgProc (WCDDP), which performs the default processing of messages for dialog boxes and the controls they contain. WinDlgBox (WCLMD) passes control to the application dialog procedure MyDlgProc. WinDlgBox returns a value, as determined by the WinDismiss dialog procedure.

If the user selects the **OK** or **Cancel** pushbutton in the dialog box, the WinDismissDlg (WCDELD) call in the dialog procedure destroys the dialog box and returns TRUE. The application calls the WinInvalidateRegion (WINVRG) function, causing the whole of the client window to be redrawn. This causes the system to generate a WM_PAINT message for the client window.

The ID_EXITPROGRAM Command Value

The **Exit Program** command value ID_EXITPROGRAM is processed by posting a WM_CLOSE message to the program.

The WM_PAINT Message

The code uses WinFillRect (WIFRCT) to draw the client window background in the current system window color. The first time through, bComplete is FALSE, so no text is drawn. Subsequently, bComplete is TRUE, and the code draws the two lines of text, using successive GpiCharStringAt (GICHAR) calls.

The WM_CLOSE Message

In the WM_COMMAND processing, the program posts itself a WM_CLOSE message when the user selects the exit choice. The program processes the WM_CLOSE message by posting itself a WM_QUIT message. When the call WinGetMsg (WMGET) finds a WM_QUIT, it returns FALSE, so the processing drops through to the termination calls.

DIALOG1 Dialog Procedure MyDlgProc

The sample dialog procedure MyDlgProc is also simple. All messages except for WM_COMMAND are handled by WinDefDlgProc (WCDDP).

When selected, the **OK** and **Cancel** pushbuttons generate WM_COMMAND messages. These are handled explicitly in the application dialog procedure.

The **OK** and **Cancel** pushbuttons generate WM_COMMAND command values of 1 and 2.

The values were given to the pushbuttons when they were created using the Dialog Box Editor. The values 1 and 2 were chosen because they are the command values generated by OS/2 when the user presses the Enter and Escape keyboard keys. 1 and 2 are defined in OS/2 as the constants DID_OK and DID_CANCEL, so the application dialog procedure tests for them.

If the command value is DID_OK, the application uses the WinQueryWindowText (WIQTXT) call to read the text from the entry field. The WinWindowFromID (WIQHID) function obtains the window handle of the entry field from its identifier value of ID_ENTRYFIELD. bComplete is a flag that was initially set to FALSE to stop the WM_PAINT processing from trying to draw any text. There is no break at the end of the DID_OK processing, so control drops through to the DID_CANCEL case, where the dialog box is dismissed with a reply value of TRUE. The reply value is returned to WinDlgBox in the window procedure. The FALSE returned at the end of the DID_CANCEL processing is processed by OS/2. It is not returned to the call that invoked the dialog procedure.

DIALOG1 Module Definition File

Here is an excerpt from DIALOG1's definition file:

```
EXPORTS                 ;

MyWindowProc              @1
MyDlgProc                 @2
```

Window procedures and dialog procedures are called by OS/2. Under the EXPORTS statement, the module definition file must declare all window procedures and dialog procedures as exported routines.

Chapter 10. The DIALOG2 Sample Application

If you read the chapters of the guide sequentially, you notice that each PM sample application shown is more complex than the previous one, but that they are all relatively short and simple. DIALOG2 is a PM sample application that provides a dialog-box-based user interface that is close to the level of complexity of a real application. Because of this, the source code of DIALOG2 is quite long. So this chapter does not list in full the source code of DIALOG2 and its associated files. You can print DIALOG2.C, DIALOG2.H, DIALOG2.RC, and DIALOG2.DLG, and use the listings in conjunction with this chapter. This chapter uses sections of code from the program to illustrate the techniques involved in programming dialog boxes and controls.

The structure of DIALOG2 consists of a main function, a window procedure for the program's main window, two window procedures for two child windows, six dialog procedures for each of the dialog boxes, and several private functions called from inside the program.

The Main Function

The main function performs tasks that you have seen in the other samples, such as initialization, creating a message queue, registering and creating the client window for the main window of the application, getting and dispatching messages, and tidying up and termination. However, in this program, the main function also registers two child windows: one is the color sample window displayed in the main window, and the other is the color preview window displayed in both the auto radio button dialog box and the check box dialog box. The second and third WinRegisterClass (WMRCL) calls in the main function register the two separate window procedures fnwpColorSample and fnwpPreview used by the child windows. The child window procedures follow the window procedure for the main window in the source listing.

Example Code: fnwpClient Window Procedure

The window procedure for the main window is called fnwpClient, and looks like this:

```
/***************** Start of first window procedure ******************/
/********************************************************************/
/*                                                                  */
/*  WinProc: fnwpClient                                             */
/*                                                                  */
/*  Controls the Client Area: loads various dialogs.                */
/*                                                                  */
/*  Side Effects:                                                   */
/*                                                                  */
/*  Modifies the values of the following globals:-                  */
/*                                                                  */
/*  szEntryField1, szEntryField2, szSelection                       */
/*                                                                  */
/********************************************************************/

MRESULT  EXPENTRY fnwpClient( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
  USHORT   Command;              /* Command passed by WM_COMMAND    */

  switch(msg)
  {
    case WM_CREATE:
      ClientCreate( hwnd );      /* Create Color Sample child       */
```

```
                                    /*  window                          */
      break;
    case WM_ERASEBACKGROUND:
      return (MRESULT)(TRUE);       /* Erase Client Area                */
    case WM_PAINT:
      ClientPaint( hwnd );          /* Invoke window painting routine   */
      break;
    case WM_HELP:
      /*****************************************************************/
      /*                                                               */
      /*  This will be received when either:-                          */
      /*                                                               */
      /*    1. The user depresses F1                                   */
      /* or 2. The user clicks on the action bar item Help             */
      /*                                                               */
      /*****************************************************************/
      Help();                       /* Invoke private Help function    */
      break;
    case WM_COMMAND:
      Command = SHORT1FROMMP( mp1 );
      ClientCommand( hwnd, Command );       /* Process the command     */
      break;
    case WM_CLOSE:
      WinPostMsg( hwnd, WM_QUIT, 0L, 0L );  /* Cause termination       */
      break;
    default:
      /* Pass all other messages to the default window procedure.      */
      Return WinDefWindowProc( hwnd, msg, mp1, mp2 );
  }
  return FALSE;
}
```

fnwpClient processes six messages, WM_CREATE, WM_ERASEBACKGROUND, WM_PAINT, WM_HELP, WM_COMMAND, and WM_CLOSE.

The WM_CREATE Message

This is processed by calling the private function ClientCreate, which calls WinCreateWindow (WICRT) to create the color sample window as a child of the client window.

The WM_ERASEBACKGROUND Message

When the frame window receives a WM_PAINT message, it sends a WM_ERASEBACKGROUND message to the client window. The application returns TRUE, which causes the frame to erase the client area.

The WM_PAINT Message

This is processed by calling the private function ClientPaint, which simply paints the text in the main window.

The WM_HELP Message

This is processed by calling the private function Help. This process is not the recommended method for providing help and will not be described in detail here. See the Template Sample Program for the recommended Help procedures.

The WM_COMMAND Message

This is processed by calling the private function ClientCommand, which processes the command values in the message. Compare this with DIALOG1, which simply nested the command value processing inside the WM_COMMAND value processing in the window procedure.

The Options Pull-Down

The following example code shows the specification of the **Options** pull-down which offers a choice of six example dialog boxes. The text and WM_COMMAND command value (given by the application-defined MI_ command values) for each choice in the pull-down are specified in the .RC file. This file was created using a text editor.

Example Code: ClientCommand Function

```
/**********************************************************************/
/*                                                                    */
/*  PRIVATE FUNCTION: ClientCommand                                   */
/*                                                                    */
/*  Take the appropriate action when a WM_COMMAND message is received*/
/*   by fnwpClient window procedure.                                  */
/*  Sets up the appropriate dialog template ID and pointer to the     */
/*   dialog procedure, for the chosen dialog type, if any.            */
/*  If exit selected, post a quit message to end program.             */
/*                                                                    */
/**********************************************************************/

VOID cdecl ClientCommand( HWND hwnd, USHORT Command )
{
  CHAR szCaption[100];
  USHORT idDlg;
  PFNWP  pfnDlgProc;


  /* Switch according to pull-down menu subitem chosen                */
  switch( Command )
  {
    case MI_EXIT:
      /* Exit selected                                                */
      WinPostMsg( hwnd, WM_QUIT, 0L, 0L );
      return;
    case MI_ABOUT:
      WinQueryWindowText ( hwndFrame, sizeof(szCaption), szCaption );
      WinMessageBox( HWND_DESKTOP,
                     WinQueryActiveWindow( HWND_DESKTOP, FALSE ),
                     (PSZ)szAbout,
                     (PSZ)szCaption,
                     ID_MB,
                     MB_MOVEABLE | MB_OK | MB_CUANOTIFICATION );
      return;
    case MI_MODELESS:
      WinShowWindow( hwndModeless, TRUE );
      WinSetFocus( HWND_DESKTOP, hwndModeless );
      return;
    case MI_ENTRYFIELD:
      /* Entry field dialog selected                                  */
      idDlg      = DLG_ENTRYFIELD;
      pfnDlgProc = (PFNWP)fnwpEntryFieldDlg;
      break;
    case MI_AUTORADIOBUTTON:
      /* Auto radio button dialog selected                            */
```

```
      idDlg      = DLG_AUTORADIOBUTTON;
      pfnDlgProc = (PFNWP)fnwpAutoRadioButtonDlg;
      break;
    case MI_CHECKBOX:
      /* Checkbox dialog selected                                    */
      idDlg      = DLG_CHECKBOX;
      pfnDlgProc = (PFNWP)fnwpCheckBoxDlg;
      break;
    case MI_LISTBOX:
      /* Listbox dialog selected                                     */
      idDlg      = DLG_LISTBOX;
      pfnDlgProc = (PFNWP)fnwpListBoxDlg;
      break;
    case MI_COMBOBOX:
      /* Combobox dialog started                                     */
      idDlg      = DLG_COMBOBOX;
      pfnDlgProc = (PFNWP)fnwpComboBoxDlg;
      break;

    default:
      /* Take no action for any other selections                     */
      return;
  }

  /* Invoke a modal dialog with the main window frame as owner       */
  WinDlgBox( HWND_DESKTOP,          /* Parent                      */
             hwndFrame,             /* Owner                       */
             pfnDlgProc,            /* Address of dialog proc      */
             NULL,                  /* Module handle               */
             idDlg,                 /* ID of dialog in resource    */
             NULL );                /* Initialization data         */

  /* Force repaint of main window                                    */
  WinInvalidateRect( hwnd, NULL, FALSE ); /* Update Client Area */
}
```

ClientCommand has a series of case statements that process each MI_ command value that can be expected as a result of the user's selections from the pull-downs. For each dialog box a separate dialog procedure exists in the application, and a corresponding dialog template is in the .DLG file (created using the Dialog Box Editor), as follows:

fnwpEntryFieldDlg
 DLG_ENTRYFIELD

fnwpAutoRadioButtonDlg
 DLG_AUTORADIOBUTTON

fnwpCheckBoxDlg
 DLG_CHECKBOX

fnwpListBoxDlg
 DLG_LISTBOX.

According to the selection from the **Dialogs** pull-down menu items, the variables idDlg and pfnDlgProc are respectively assigned the correct dialog template identifier and dialog procedure address for the chosen example dialog. For example, for the entry field dialog:

```
idDlg      = DLG_ENTRYFIELD;
pfnDlgProc = (PFNWP)fnwpEntryFieldDlg;
```

idDlg and pfnDlgProc are used as input parameters to WinDlgBox (WCLMD), which invokes a modal dialog with the appropriate template and procedure. Each of the four dialog procedures contain case statements that process the user's interaction with the associated dialog box. Each dialog procedure removes its dialog box from the screen when the

user has finished with it. After WinDlgBox, therefore, the main window contains a rectangular "hole", requiring redrawing. WinDlgBox (WCLMD) is followed by a WinInvalidateRect (WINVR) call to force a repaint of the main window, to fill in the hole.

The WM_CLOSE Message

This is processed by the application posting itself a WM_QUIT message.

The Dialog Procedures

All six dialog procedures start with a case statement for WM_INITDLG. This message is sent to the dialog procedure by WinDlgBox when the dialog box is created. Each dialog procedure processes the message by:

- Calling the private function CenterDlgBox.

 This function uses WinQuerySysValue (WMQSYS) to query the width and height of the display, and positions the dialog box in the center of the screen.

- Calling the private function SetSystemMenu.

 This function gets the handle of the system menu pulldown, then removes all items from the system menu pulldown that are not needed by the application.

All six dialog procedures also have a case statement for WM_HELP. In the template of each dialog box, the **Help** pushbutton is defined with a button style of BS_HELP. This tells the system to post a WM_HELP message to the application message queue when the user clicks on **Help** in each dialog box. The application dispatches the message to the dialog procedure. Each dialog procedure responds by calling the Help private function, which displays a message box.

Note: This is not the recommended procedure for providing help. See the Template Sample Program for the recommended Help procedures.

All the dialog procedures end by using WinDismissDlg (WCDELD) to remove the dialog box from the screen. They also all pass unprocessed messages to WinDefDlgProc.

Example Code: Entry Field Dialog Template

```
DLGTEMPLATE DLG_ENTRYFIELD LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Entry Field Example", DLG_ENTRYFIELD, 7, 40, 146, 112,
           WS_VISIBLE | WS_CLIPSIBLINGS | WS_SAVEBITS,
           FCF_DLGBORDER | FCF_NOBYTEALIGN | FCF_TITLEBAR | FCF_SYSMENU
    BEGIN
        CONTROL "Enter data into the two entry ", ID_NULL, 6, 100, 124, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "fields below and press ENTER.", ID_NULL, 6, 92, 136, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "The program will then verify ", ID_NULL, 6, 78, 118, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "that fields contain integer", ID_NULL, 6, 70, 118, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "values in the range 1-100.", ID_NULL, 6, 62, 118, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "Entry Field 1 ...", ID_NULL, 20, 40, 70, 10, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "Entry Field 2 ...", ID_NULL, 20, 28, 70, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "", EF_1, 92, 42, 32, 8, WC_ENTRYFIELD, ES_LEFT | ES_AUTOSCROLL |
                ES_MARGIN | WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "", EF_2, 92, 28, 32, 8, WC_ENTRYFIELD, ES_LEFT | ES_AUTOSCROLL |
                ES_MARGIN | WS_TABSTOP | WS_VISIBLE
```

```
        CONTROL "Enter", 1, 6, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON | BS_DEFAULT |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", 2, 54, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_VISIBLE
    END
END
```

The entry field example dialog box asks the user to enter two integer values in the range 1 through 100. When the user selects **OK** the numbers are validated. If they are correct, the dialog box disappears, and the numbers are displayed in the main window. If they are not correct, the application displays a message box. When the user dismisses the message box, the cursor appears in the erroneous entry field. The entry fields have the symbolic identifiers EF_1 and EF_2. The values of the identifiers are specified in the .H file.

Example Code: fnwpEntryFieldDlg Entry Field Dialog Procedure

```
/****************** Start of third dialog procedure ******************/
/*********************************************************************/
/*                                                                   */
/* DlgProc:  fnwpEntryFieldDlg                                       */
/*                                                                   */
/* A dialog procedure that captures and validates the contents of    */
/* two entry fields.                                                 */
/*                                                                   */
/* Side effects.                                                     */
/*                                                                   */
/* Modifies the globals szEntryField1, szEntryField2,                */
/*  to save the entered values.                                      */
/*                                                                   */
/*********************************************************************/

MRESULT EXPENTRY fnwpEntryFieldDlg( HWND hwndDlg, USHORT msg, MPARAM mp1,
                                    MPARAM mp2 )
{
  switch (msg)
  {
    case WM_INITDLG:
      SetSystemMenu( hwndDlg );   /* Edit dialog system menu         */
      CentreDlgBox( hwndDlg );    /* Position centrally              */
      break;
    case WM_HELP:
      Help();                     /* Invoke private Help function    */
      break;
    case WM_COMMAND:
      switch( SHORT1FROMMP( mp1 ) )
      {
        case DID_OK:    /* Enter key or pushbutton pressed/ selected */

          /*************************************************************/
          /*                                                           */
          /*  Validate the contents of the two entry fields, and       */
          /*   if either entry field is not valid, return leaving      */
          /*  dialog box visible (by not dismissing it)                */
```

```
            /*                                                    */
            /******************************************************/

            if( !IsIntInRange( hwndDlg, EF_1, 1, 100, ERR_EFINVALID, ID_MB ) )
              return FALSE;
            if( !IsIntInRange( hwndDlg, EF_2, 1, 100, ERR_EFINVALID, ID_MB ) )
              return FALSE;

            /******************************************************/
            /*                                                    */
            /* Both entry fields are valid - save their contents. */
            /* (A modal dialog is destroyed before control is returned */
            /* to the invoking winproc, so it must pass the contents   */
            /* of its Entry Fields etc back to the invoking window     */
            /* before it returns.)                                */
            /*                                                    */
            /******************************************************/

            WinQueryWindowText( WinWindowFromID( hwndDlg, EF_1 ),
                                sizeof( szEntryField1),
                                szEntryField1 );
            WinQueryWindowText( WinWindowFromID( hwndDlg, EF_2 ),
                                sizeof( szEntryField2),
                                szEntryField2 );

          case DID_CANCEL:/* Escape key or CANCEL pushbutton pressed/selected */
            WinDismissDlg( hwndDlg,TRUE ); /* Finished with dialog box */
            break;
          default:
            break;
        }
        break;

      default:  /* Pass all other messages to the default dialog proc  */
        return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
    }
    return FALSE;
}

/****************** End of third dialog procedure ********************/
```

The WM_COMMAND Message

The application is interested in the contents of the entry fields only when the user selects **OK**. At this point, a WM_COMMAND message is posted to the application. It contains the command value 1, defined in the OS/2 header files as DID_OK. Inside the WM_COMMAND processing, a case statement for DID_OK calls the private function IsIntInRange to validate the contents of each of the two entry fields.

Because the contents of each entry field are in ASCII text, IsIntInRange uses the WinQueryDlgItemShort (WCQDIS) call to convert the text to binary, and tests that it is in the range 1 through 100. If one of the numbers is not within the range, a message box is displayed. When the user dismisses the message box, WinSetFocus (WMSFOC) is used to set the input focus (and therefore the cursor) to the field in error. If both integers are in the correct range, WinQueryWindowText (WIQTXT) is used to copy the contents of each into buffers. WinQueryWindowText requires as an input parameter the window handle of the window containing the window text. WinWindowFromID (WIQHID) is used to find out the window handle of a child window (in this case, each entry field) given its symbolic identifier and parent window handle.

Example Code: Auto Radio Button Dialog Template

```
DLGTEMPLATE DLG_AUTORADIOBUTTON LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Auto Radio Button Example", DLG_AUTORADIOBUTTON, 3, 45, 133, 122,
           WS_VISIBLE | WS_CLIPSIBLINGS | WS_SAVEBITS,
           FCF_DLGBORDER ] FCF_NOBYTEALIGN ] FCF_TITLEBAR | FCF_SYSMENU
    BEGIN
        CONTROL "Select one of the radio", ID_NULL, 12, 108, 101, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "buttons, and the color", ID_NULL, 12, 100, 101, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "sample window will be", ID_NULL, 12, 92, 102, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "repainted in the color of", ID_NULL, 12, 84, 109, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "your choice.", ID_NULL, 12, 76, 54, 8, WC_STATIC, SS_TEXT |
                DT_CENTER | DT_TOP | WS_VISIBLE
        WINDOW  "", RECTL_AUTORADIOBUTTON, 73, 37, 32, 30, "Preview",
                WS_VISIBLE
        CONTROL "Color Preview", ID_NULL, 58, 25, 63, 8, WC_STATIC, SS_TEXT |
                DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "~Red", RB_RED, 6, 54, 38, 11, WC_BUTTON, BS_AUTORADIOBUTTON |
                WS_TABSTOP | WS_GROUP | WS_VISIBLE
        CONTROL "~Green", RB_GREEN, 6, 44, 41, 10, WC_BUTTON, BS_AUTORADIOBUTTON |
                WS_VISIBLE
        CONTROL "~Blue", RB_BLUE, 6, 34, 38, 10, WC_BUTTON, BS_AUTORADIOBUTTON |
                WS_VISIBLE
        CONTROL "OK", 1, 6, 4, 36, 12, WC_BUTTON, BS_PUSHBUTTON | BS_DEFAULT |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", 2, 45, 4, 36, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_VISIBLE
    END
END
```

The auto radio button dialog box displays a preview window and three radio buttons, which set the colors red, green, or blue. The user cab select a single color, which is shown in the preview window. When the user selects the **OK** pushbutton, the dialog box disappears, and the color of the sample window in the main window changes to the selected color.

Example Code: fnwpAutoRadioButtonDlg Radio Button Dialog Procedure

```
/****************** Start of first dialog procedure ******************/
/*********************************************************************/
/*                                                                   */
/* DlgProc:  fnwpAutoRadioButtonDlg                                  */
/*                                                                   */
/* A dialog procedure that uses auto radio buttons to change the     */
/*  color of the Preview window and the Color Sample window.         */
/* One color only can therefore be selected.                         */
/*                                                                   */
/* Side effects:                                                     */
/*                                                                   */
/* Modifies the Color Sample window color via the global clrSample   */
```

```
/*                                                                    */
/**********************************************************************/

MRESULT EXPENTRY fnwpAutoRadioButtonDlg( HWND hwndDlg, USHORT msg,
                                         MPARAM mp1, MPARAM mp2 )
{
  switch (msg)
  {
    case WM_INITDLG:
      SetSystemMenu( hwndDlg );   /* Edit dialog screen             */
      CentreDlgBox( hwndDlg );    /* Position centrally on screen   */
      break;
    case WM_HELP:
      Help();                     /* Invoke the private Help function*/
      break;
    case WM_CONTROL:
      if( SHORT2FROMMP( mp1 ) == BN_CLICKED )
        /* Determine Preview window new color from the selected     */
        /*  auto radio button                                       */
        switch( SHORT1FROMMP( mp1 ) )
        {
          case RB_RED:
            clrPreview = CLR_RED;
            break;
          case RB_GREEN:
            clrPreview = CLR_GREEN;
            break;
          case RB_BLUE:
            clrPreview = CLR_BLUE;
            break;
          default:
           return FALSE;
        }
        /* Force repaint of Preview window                          */
        WinInvalidateRect( WinWindowFromID( hwndDlg, RECTL_AUTORADIOBUTTON ),
                           NULL, FALSE );
        break;
    case WM_COMMAND:
      switch( SHORT1FROMMP( mp1 ) )
      {
        case DID_OK:    /* Enter key or pushbutton pressed/ selected */
          /* Force repaint of Color Sample window                   */
          clrSample = clrPreview; /* Set color to Preview color     */
          WinInvalidateRect( hwndColorSample, NULL, FALSE );
          break;
        case DID_CANCEL: /* Escape key or CANCEL pushbutton pressed/selected */
          break;
        default:
          return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
      }
      WinDismissDlg( hwndDlg, TRUE ); /* Finished with dialog box   */
      break;

    default:  /* Pass all other messages to the default dialog proc */
      return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
  }
  return FALSE;
}
```

The WM_INITDLG Message

In addition to centering the dialog box and setting up the dialog system menu, the processing of WM_INITDLG in this procedure calls WinInvalidateRect to cause the preview window to be painted in the same color as the sample window in the main window. It does this because the sample window color may have been changed by the user interacting with the check box example dialog box. WinWindowFromID is used to find out the preview window handle from its symbolic identifier RECTL_AUTORADIOBUTTON. The program does not explicitly have to create the preview window. It is drawn by the system, because it is specified as a child window of the dialog box in the .DLG file. Compare this with the color sample window, which DIALOG2 explicitly created, using WinCreateWindow.

The color preview windows in the auto radio button and check box example dialog boxes were added to the .DLG files using a text editor.

The WM_CONTROL Message

The system checks and unchecks BS_AUTORADIOBUTTON style buttons. You do not have to worry about checking and unchecking them in your application. However, the fnwpAutoRadioButtonDlg procedure monitors user interaction with the radio buttons *before* **OK** is selected. The procedure changes the color of the preview window, as a result of the state of the radio buttons, while the dialog box is still displayed. (Compare this with the entry field example dialog procedure, which acts on the contents of the dialog box only when the user selects **OK**, as indicated by the WM_COMMAND message.) When the user clicks on an auto radio button, the system paints the button black to show it has been selected, deselects the other radio buttons, and sends a WM_CONTROL message to the application. The notification code is BN_CLICKED. The first parameter of the message contains the identifier of the radio button. The identifier is RB_RED, RB_GREEN, or RB_BLUE, as specified in the .DLG file. The WM_CONTROL processing tests for these, sets the color of the preview window, and calls WinInvalidateRect to repaint the preview window.

The WM_COMMAND Message

In the WM_COMMAND processing, when the user selects **OK**, as indicated by DID_OK, the color of the sample window is set to the same color as the preview window, and is repainted.

Example Code: Check Box Dialog Template

```
DLGTEMPLATE DLG_CHECKBOX LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Check Box Example", DLG_CHECKBOX, 3, 38, 138, 115,
           WS_VISIBLE | WS_CLIPSIBLINGS | WS_SAVEBITS,
           FCF_DLGBORDER ] FCF_NOBYTEALIGN ] FCF_TITLEBAR | FCF_SYSMENU
    BEGIN
        CONTROL "Select one or more check", ID_NULL, 6, 101, 122, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "boxes, and the color sample", ID_NULL, 6, 93, 123, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "window will be painted in the", ID_NULL, 6, 85, 128, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "color(s) of your choice.", ID_NULL, 6, 77, 103, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        WINDOW  "", RECTL_CHECKBOX, 70, 36, 38, 28, "Preview", WS_VISIBLE
        CONTROL "Color Preview", ID_NULL, 58, 24, 63, 8, WC_STATIC, SS_TEXT |
                DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "~Red", CB_RED, 6, 55, 42, 10, WC_BUTTON, BS_CHECKBOX |
                WS_TABSTOP | WS_GROUP | WS_VISIBLE
```

```
        CONTROL "~Green", CB_GREEN, 6, 45, 42, 10, WC_BUTTON, BS_CHECKBOX |
                WS_VISIBLE
        CONTROL "~Blue", CB_BLUE, 6, 35, 42, 10, WC_BUTTON, BS_CHECKBOX |
                WS_VISIBLE
        CONTROL "OK", 1, 6, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON | BS_DEFAULT |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", 2, 50, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_VISIBLE
    END
END
```

The check box dialog box displays a preview window and three check boxes, corresponding to the colors red, green, and blue. The user can select one or more colors, which are shown, mixed if necessary, in the preview window. When the user clicks the **OK** pushbutton, the dialog box disappears, and the color of the sample window in the main window changes to the selected color or color mix. The example uses ordinary check boxes, not auto check boxes. The application is responsible, therefore, for checking or unchecking the boxes in response to the user's interaction.

Example Code: fnwpCheckBoxDlg Check Box Dialog Procedure

```
/***************** Start of second dialog procedure *****************/
/********************************************************************/
/*                                                                  */
/* DlgProc:  fnwpCheckBoxDlg                                        */
/*                                                                  */
/* A dialog procedure that uses check boxes to change the color     */
/* of the Preview window and the Color Sample window.               */
/*  One or more color(s) can therefore be selected.                 */
/*                                                                  */
/* Side effects:                                                    */
/*                                                                  */
/* Modifies the Color Sample window color via the global clrSample  */
/*                                                                  */
/********************************************************************/

MRESULT EXPENTRY fnwpCheckBoxDlg( HWND hwndDlg, USHORT msg, MPARAM mp1,
                                  MPARAM mp2 )
{
  switch (msg)
  {
    case WM_INITDLG:
      SetSystemMenu( hwndDlg );   /* Edit dialog system menu         */
      CentreDlgBox( hwndDlg );    /* Position centrally              */

      /* Determine current color components of clrSample, and set    */
      /*  check boxes accordingly (so that Preview window is set     */
      /*  initially to same color as Sample window )                 */
      clrPreview = clrSample;
      if( (clrPreview & CLR_RED) == CLR_RED )
        WinPostMsg( WinWindowFromID( hwndDlg, CB_RED ),
                    BM_SETCHECK,
                    MPFROM2SHORT( TRUE,0 ),
                    0L );
      if( (clrPreview & CLR_GREEN) == CLR_GREEN )
        WinPostMsg( WinWindowFromID( hwndDlg, CB_GREEN ),
```

```
                    BM_SETCHECK,
                    MPFROM2SHORT( TRUE,0 ),
                    0L );
        if( (clrPreview & CLR_BLUE) == CLR_BLUE )
          WinPostMsg( WinWindowFromID( hwndDlg, CB_BLUE ),
                    BM_SETCHECK,
                    MPFROM2SHORT( TRUE,0 ),
                    0L );

        /* Force initial paint of Preview window in the color(s) set   */
        /*  above                                                      */
        WinInvalidateRect( WinWindowFromID( hwndDlg, RECTL_CHECKBOX ),
                           NULL, FALSE );
        break;
      case WM_HELP:
        Help();                      /* Invoke private Help function    */
        break;
      case WM_CONTROL:               /* User has clicked on a check box */
        if( SHORT2FROMMP( mp1 ) == BN_CLICKED )
        {
          /* This may be a checkbox selection or deselection -         */
          /*  add or subtract the latest color change                  */
          CheckColor( hwndDlg, SHORT1FROMMP( mp1 ) );

          /* Force repaint of preview window in the new color(s)       */
          WinInvalidateRect( WinWindowFromID( hwndDlg, RECTL_CHECKBOX ),
                             NULL, FALSE );
        }
        break;
      case WM_COMMAND:
        switch( SHORT1FROMMP( mp1 ) )
        {
          case DID_OK:    /* Enter key or pushbutton pressed/ selected */
            /* Force repaint of color sample window in new color(s)    */
            clrSample = clrPreview;
            WinInvalidateRect( hwndColorSample, NULL, FALSE );
            break;
          case DID_CANCEL: /* Escape key or CANCEL pushbutton pressed/selected */
            break;
          default:
            return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
        }
        WinDismissDlg( hwndDlg, TRUE ); /* Finished with dialog box    */
        return FALSE;

      default:  /* Pass all other messages to the default dialog proc  */
        return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
    }
    return FALSE;
}

/****************** End of second dialog procedure ******************/
```

The WM_INITDLG Message

In addition to centering the dialog box and setting up the dialog system menu, the processing of WM_INITDLG in this procedure determines the component colors of the color of the sample window. It also ensures that the status of the check boxes reflects the colors, and paints the preview window in the same color as the sample window. The procedure does this because the sample window color may have been changed by the user interacting with the auto radio button example dialog box. The appropriate check boxes are set by posting the message BM_SETCHECK to the control.

The WM_CONTROL Message

When the user selects or deselects a check box, the system sends a WM_CONTROL message to the application. The notification code is BN_CLICKED. The first parameter of the message contains the identifier of the check box. The identifier is CB_RED, CB_GREEN, or CB_BLUE, as specified in the .DLG file. The private function CheckColor is called.

The WM_COMMAND Message

The **OK** case of DID_OK is processed by setting the sample window color to that of the preview window, and forcing a repaint. The **Cancel** case does nothing. Both cases break out to the WinDismissDlg.

Example Code: CheckColor Private Function

```
/**************************************************************************/
/*                                                                        */
/* PRIVATE FUNCTION: CheckColor                                           */
/*                                                                        */
/* Toggle the Checked/UnChecked state of a check box and add/remove       */
/* the corresponding CLR color component of the Preview window color.     */
/*                                                                        */
/* Modifies the global clrPreview.                                        */
/*                                                                        */
/**************************************************************************/

VOID cdecl CheckColor( HWND hwndDlg, SHORT idDlgItem )
{
  BOOL  fChecked;
  COLOR clr;

  switch( idDlgItem )
  {
    /* Set color according to check box id value                          */
    case CB_RED:
      clr = CLR_RED;
      break;
    case CB_GREEN:
      clr = CLR_GREEN;
      break;
    case CB_BLUE:
      clr = CLR_BLUE;
      break;
    default:
      return;
  }

  /* Query the checked or unchecked state of the check box, and           */
```

```
  /*  set fChecked true (checked) or false (unchecked) accordingly   */
  fChecked = SHORT1FROMMR( WinSendMsg( WinWindowFromID( hwndDlg , idDlgItem ),
                                       BM_QUERYCHECK,
                                       0L,
                                       0L ) );

  /* Set or reset the check box state, to the reverse of fChecked     */
  WinPostMsg( WinWindowFromID( hwndDlg, idDlgItem ),
              BM_SETCHECK,
              MPFROM2SHORT( !fChecked, 0 ),
              0L );

  /* Add or subtract the checked or unchecked color choice           */
  if( fChecked )                      /* If color previously checked */
    clrPreview -= clr;                /* subtract it ... else        */
  else
    clrPreview += clr;                /* ... add it.                 */
}
```

To find out which check box was clicked on, CheckColor tests for the identifier CB_RED, CB_GREEN, or CB_BLUE, and sets the variable clr to the appropriate system color. The application knows only that the user has clicked on the red, green, or blue check boxes at this point. Remember that because they are *not* auto check boxes, the system has not checked them, so the application must do it. WinSendMsg (WMSEND) is used to send a BM_QUERYCHECK message to the checked or unchecked check box, to determine its state before the user clicked on it. The button control procedure responds in the message. The SHORT1FROMMR macro extracts the reply, which is TRUE or FALSE, and it is placed in fChecked. fChecked is the status of the control before the user clicked on it. The application sets the control to the opposite of fChecked. This is done by using WinPostMsg (WMPOST) to post a BM_SETCHECK message to the check box, with fChecked in the first parameter. CheckColor adds or subtracts the selected or deselected color from the preview window color. Back in fnwpCheckBoxDlg, a WinInvalidateRect forces a repaint of the color preview window.

Example Code: List Box Dialog Template

```
DLGTEMPLATE DLG_LISTBOX LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "List Box Example", DLG_LISTBOX, 5, 36, 146, 105,
           WS_VISIBLE | WS_CLIPSIBLINGS | WS_SAVEBITS,
           FCF_DLGBORDER ] FCF_NOBYTEALIGN ] FCF_TITLEBAR
    BEGIN
        CONTROL "Select an item and press Enter", ID_NULL, 6, 93, 139, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "", LISTBOX, 25, 26, 98, 58, WC_LISTBOX, WS_TABSTOP | WS_VISIBLE
        CONTROL "Enter", 1, 6, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON | BS_DEFAULT |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", 2, 54, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_VISIBLE
    END
END
```

The list box dialog box displays a list of text strings, and allows a single selection from the list. When the user selects **OK**, the selected string is displayed in the main window.

Example Code: fnwpListBoxDlg List Box Dialog Procedure

```
/****************** Start of fourth dialog procedure ******************/
/**********************************************************************/
/*                                                                    */
/* DlgProc:  fnwpListBoxDlg                                           */
/*                                                                    */
/* A dialog procedure that displays a list of text strings and      */
/*  permits a single selection from the list                          */
/*                                                                    */
/* Side effects:                                                      */
/*                                                                    */
/* Modifies the global szSelection,                                   */
/*  to save the contents of a selected list box item                   */
/*                                                                    */
/**********************************************************************/

MRESULT EXPENTRY fnwpListBoxDlg( HWND hwndDlg, USHORT msg, MPARAM mp1,
                                 MPARAM mp2 )
{
  CHAR szBuffer[LEN_LISTBOXENTRY];
  INT    i;
  SHORT  id;

  switch (msg)
  {
    case WM_INITDLG:
      SetSystemMenu( hwndDlg );   /* Edit dialog system menu         */
      CentreDlgBox( hwndDlg );    /* Position centrally              */

      /****************************************************************/
      /*                                                              */
      /*  Initialize the list box with a set of strings loaded from a */
      /*  resource file.                                              */
      /*                                                              */
      /****************************************************************/

      for ( i = 0; i < NUM_LISTBOXENTRIES; i++ )
      {
        /* Load from stringtable into an intermediate variable       */
        WinLoadString( hab,
                       NULL,
                       LBI_1 + i,
                       LEN_LISTBOXENTRY,
                       (PSZ)szBuffer
                     );
        /* Add item to the list box                                   */
        WinSendDlgItemMsg( hwndDlg,
                           LISTBOX,
                           LM_INSERTITEM,
                           MPFROM2SHORT( LIT_END, 0 ),
                           MPFROMP( szBuffer )
```

```
                    );
  }

  /* Make the first list item initially selected                */
  WinSendDlgItemMsg( hwndDlg,
                     LISTBOX,
                     LM_SELECTITEM,
                     MPFROMSHORT( 0 ), /* First item index value */
                     MPFROMSHORT( TRUE  )
                   );
  break;
case WM_HELP:
  Help();
  break;
case WM_CONTROL:
  switch( SHORT2FROMMP( mp1 ) )
  {
    case LN_ENTER:                  /* Catch double click on */
      WinPostMsg( hwndDlg,          /* list box item, and    */
                  WM_COMMAND,       /* simulate Enter        */
                  MPFROM2SHORT( DID_OK, CMDSRC_OTHER ),
                  MPFROMLONG( 1L )  /* pushbutton selection  */
                );
      break;
    default:
      break;
  }
  break;
case WM_COMMAND:
  switch( SHORT1FROMMP( mp1 ) )
  {
    case DID_OK:    /* Enter key or pushbutton pressed/ selected */
      /*************************************************************/
      /*                                                           */
      /* Find out whether any item was selected. If yes, return    */
      /*  selected item text.                                      */
      /* (By default, only one item can be selected)               */
      /*                                                           */
      /*************************************************************/

      id = SHORT1FROMMR( WinSendDlgItemMsg( hwndDlg,
                                            LISTBOX,
                                            LM_QUERYSELECTION,
                                            0L,
                                            0L ) );
      if( id == LIT_NONE )
        strcpy( szSelection, "" ); /* Nothing selected          */
      else
        WinSendDlgItemMsg( hwndDlg,
                           LISTBOX,
                           LM_QUERYITEMTEXT,
                           MPFROM2SHORT( id, LEN_LISTBOXENTRY ),
                           MPFROMP( szSelection ) );
    case DID_CANCEL: /* Escape key or CANCEL pushbutton pressed/selected */
      WinDismissDlg( hwndDlg, TRUE ); /* Finished with dialog box*/
      return FALSE;
    default:
      break;
  }
  break;
```

```
    default:  /* Pass all other messages to the default dialog proc */
      return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
  }
  return FALSE;
}

/***************** End of fourth dialog procedure *******************/
```

The WM_INITDLG Message

In addition to centering the dialog box and setting up the dialog system menu, WM_INITDLG collects and loads strings from a resource file into the list box. The order of the strings in the list box is determined by their order in the resource file.

When all the strings have been loaded, WM_INITDLG sends a message using the WinSendDlgItemMsg (WCSDIM) call to select the first item in the list. This is in accordance with the Common User Access requirements, which require an initial selection to be shown when a list box is displayed.

The WM_CONTROL Message

WM_CONTROL, with a notification code of LN_ENTER, is received when the user double-clicks on an item in the list box. The message is processed by posting a WM_COMMAND message to simulate the user selecting the **OK** pushbutton or pressing the Enter key.

The WM_COMMAND Message

The list box example is similar to the entry field example, in that its dialog procedure is not interested in the user's interaction with the list. Like the entry field example, the procedure is interested in the selection state only when the user presses **OK**.

The DID_OK case is processed by sending an LM_QUERYSELECTION message to the list box. The message returns the index of the selected item, or LIT_NONE if no item is selected, in the ID variable. If a selection has been made, the procedure sends an LM_QUERYITEM TEXT message to the list box, to copy the text of the selected item into szSelection. The DID_OK processing falls through to the DID_CANCEL processing, which dismisses the dialog box.

Example Code: fnwpComboBoxDlg Combobox Dialog Procedure

```
/***************** Start of fifth dialog procedure ******************/
/********************************************************************/
/*                                                                  */
/* DlgProc:  fnwpComboBoxDlg                                        */
/*                                                                  */
/* A dialog procedure that displays a list of text strings in a     */
/* Combo box and permits a single selection from the list, or an    */
/* entry in the edit box.                                           */
/*                                                                  */
/* Side effects                                                     */
/*                                                                  */
/* Modifies the global szComboSel,                                  */
/*  to save the contents of the ComboBox edit field                 */
/*                                                                  */
/********************************************************************/
```

```
MRESULT EXPENTRY fnwpComboBoxDlg( HWND hwndDlg, USHORT msg, MPARAM mp1,
                                  MPARAM mp2 )
{
  CHAR szBuffer[LEN_COMBOBOXENTRY];
  INT    i;

  switch (msg)
  {
    case WM_INITDLG:
      SetSystemMenu( hwndDlg );   /* Edit dialog system menu        */
      CentreDlgBox( hwndDlg );    /* Position centrally             */

      /*****************************************************************/
      /*                                                               */
      /*  Initialize the combobox with a set of strings loaded from  */
      /*  a resource file.                                             */
      /*                                                               */
      /*****************************************************************/

      for ( i = 0; i < NUM_LISTBOXENTRIES; i++ )
      {
        /* Load from stringtable into an intermediate variable       */
        WinLoadString( hab,
                       NULL,
                       LBI_1 + i,
                       LEN_COMBOBOXENTRY,
                       (PSZ)szBuffer
                     );
        /* Add item to the combobox                                  */
        WinSendDlgItemMsg( hwndDlg,
                           COMBOBOX,
                           LM_INSERTITEM,
                           MPFROM2SHORT( LIT_END, 0 ),
                           MPFROMP( szBuffer )
                         );
      }

      /*****************************************************************/
      /*                                                               */
      /*  Set the text limit of the edit field of the Combo box.      */
      /*                                                               */
      /*****************************************************************/
      WinSendDlgItemMsg( hwndDlg,
                         COMBOBOX,
                         EM_SETTEXTLIMIT,
                         MPFROMSHORT( LEN_COMBOBOXENTRY ),
                         (MPARAM)NULL );

      break;
    case WM_HELP:
      Help();
      break;
    case WM_CONTROL:
      switch( SHORT2FROMMP( mp1 ) )
      {
        case CBN_ENTER:                 /* Catch double click on */
          WinPostMsg( hwndDlg,          /* list box item, and    */
                      WM_COMMAND,       /* simulate Enter        */
                      MPFROM2SHORT( DID_OK, CMDSRC_OTHER ),
```

```
                    MPFROMLONG( 1L )    /* pushbutton selection  */
                  );
            break;
          default:
            break;
        }
        break;
      case WM_COMMAND:
        switch( SHORT1FROMMP( mp1 ) )
        {
          case DID_OK:    /* Enter key or pushbutton pressed/ selected */
            /*************************************************************/
            /*                                                           */
            /* Store the text, if any, entered in the combo box edit     */
            /*  field.                                                   */
            /*                                                           */
            /*************************************************************/

            WinQueryWindowText( WinWindowFromID( hwndDlg,
                                                 COMBOBOX),
                                LEN_COMBOBOXENTRY,
                                szComboSel );

          case DID_CANCEL: /* Escape key or CANCEL pushbutton pressed/selected */
            WinDismissDlg( hwndDlg, TRUE ); /* Finished with dialog box*/
            return FALSE;
          default:
            break;
        }
        break;

      default:  /* Pass all other messages to the default dialog proc */
        return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
    }
    return FALSE;
}

/***************** End of fifth dialog procedure *********************/
```

The WM_INITDLG Message

This message performs the same function as in the list box dialog procedure. When all the strings are in the combobox, a further call sets the text limit for the edit field of the combobox.

The WM_CONTROL Message

The processing of this message is the same as in the list box dialog procedure.

The WM_COMMAND Message

The application is interested in the text that the user has typed into the edit field only when the user presses Enter. The DID_OK case is processed by calling WinQueryWindowText with the window handle of the combobox.

Example Code: fnwpModelessDlg Modeless Dialog Procedure

```
/**************** Start of modeless dialog procedure ****************/
/********************************************************************/
/*                                                                  */
/* DlgProc:  fnwpModelessDlg                                        */
/*                                                                  */
/* A dialog procedure that displays a list of text colors and       */
/*  permits a single selection from the list                        */
/*                                                                  */
/* Side effects                                                     */
/*                                                                  */
/* Modifies the global clrText,                                     */
/*  to save the new text color and forces a repaint of the client   */
/*  area                                                            */
/********************************************************************/

MRESULT EXPENTRY fnwpModelessDlg( HWND hwndDlg, USHORT msg, MPARAM mp1,
                                  MPARAM mp2 )
{
  CHAR szBuffer[LEN_COLORTEXT];
  INT    i;
  SHORT  id;
  PSWP   pswp;

  switch (msg)
  {
    case WM_INITDLG:
      SetSystemMenu( hwndDlg );   /* Edit dialog system menu        */

      /****************************************************************/
      /*                                                              */
      /*  Initialize the listbox with a set of strings loaded from a */
      /*  resource file.                                              */
      /*                                                              */
      /****************************************************************/

      for ( i = 0; i < NUM_COLORS; i++ )
      {
        /* Load from stringtable into an intermediate variable       */
        WinLoadString( hab,
                       NULL,
                       LBC_1 + i,
                       LEN_LISTBOXENTRY,
                       (PSZ)szBuffer
                     );
        /* Add item to the listbox                                   */
        WinSendDlgItemMsg( hwndDlg,
                           LISTBOX,
                           LM_INSERTITEM,
                           MPFROM2SHORT( LIT_END, 0 ),
                           MPFROMP( szBuffer )
                         );
      }
```

```
  /* Make the default color initially selected                    */

  WinSendDlgItemMsg( hwndDlg,
                     LISTBOX,
                     LM_SELECTITEM,
                     MPFROMSHORT( clrText - 1 ),
                     MPFROMSHORT( TRUE  )
                   );
  break;
case WM_HELP:
  Help();
  break;
case WM_ADJUSTWINDOWPOS:
  pswp = (PSWP)mp1;
  if( pswp->fs & SWP_SHOW )
     CentreDlgBox( hwndDlg );         /* Position centrally    */
  return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
case WM_CONTROL:
  switch( SHORT2FROMMP( mp1 ) )
  {
    case LN_ENTER:                    /* Catch double click on */
      WinPostMsg( hwndDlg,            /* list box item, and    */
                  WM_COMMAND,         /* simulate Enter        */
                  MPFROM2SHORT( DID_OK, CMDSRC_OTHER ),
                  MPFROMLONG( 1L )    /* pushbutton selection  */
                );
      break;
    default:
      break;
  }
  break;
case WM_COMMAND:
  switch( SHORT1FROMMP( mp1 ) )
  {
    case DID_OK:   /* Enter key or pushbutton pressed/ selected */
      /*************************************************************/
      /*                                                           */
      /* Find out which item is selected and change the client     */
      /*  area text to that color.                                 */
      /* (By default, only one item can be selected)               */
      /*                                                           */
      /*************************************************************/

      id = SHORT1FROMMR( WinSendDlgItemMsg( hwndDlg,
                                            LISTBOX,
                                            LM_QUERYSELECTION,
                                            0L,
                                            0L ) );
      if( id != LIT_NONE )
      {
         clrSaveText = clrText;
         clrText = id + 1;
         WinInvalidateRect( hwndClient, NULL, FALSE );
      }
      return FALSE;
    case DID_CANCEL: /* Escape key or CANCEL pushbutton pressed/selected */
      WinDismissDlg( hwndDlg, TRUE ); /* Finished with dialog box*/
      return FALSE;
    case UNDO:
      clrText = clrSaveText;
```

```
            WinInvalidateRect( hwndClient, NULL, FALSE );
            return FALSE;
          default:
            break;
        }
        break;

      default:  /* Pass all other messages to the default dialog proc */
        return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
    }
    return FALSE;
}

/***************** End of modeless dialog procedure ******************/
```

The WM_INITDLG Message

This message performs the same function as in the list box dialog procedure, except that it does not center the dialog box. When all the strings are in the list box, a further call to WinSendDlgItemMsg (WCSDIM) sends an LM_SELECTITEM message to select the CLR_NEUTRAL item.

The WM_ADJUSTWINDOWPOS Message

This message centers the dialog box in the display each time it is called up.

The WM_CONTROL Message

The processing of this message is the same as in the list box dialog procedure.

The WM_COMMAND Message

The modeless example is similar to the entry field and list box examples, in that its dialog procedure is not interested in the user's interaction with the list. Like the other examples, the procedure is interested in the selection state only when the user presses **OK**.

The DID_OK case is processed by sending an LM_QUERYSELECTION message to the list box. The message returns the index of the selected item (it is the name of a color) and forces the window to be repainted.

Chapter 11. The DLG1 Nonreentrant Sample Application

The following nonreentrant sample applications are provided in the OS/2 1.2 Programming Tools:

DLG1COB
COBOL sample application using a single dialog box

DLG2COB
COBOL sample application using several dialog boxes and controls

DLG1FOR
FORTRAN sample application using a single dialog box

DLG2FOR
FORTRAN sample application using several dialog boxes and controls.

There is a separate directory for each application. The directories are:

```
TOOLKT12\COBOL\SAMPLES\PM\DLG1COB
TOOLKT12\COBOL\SAMPLES\PM\DLG2COB
TOOLKT12\FORTRAN\SAMPLES\PM\DLG1FOR
TOOLKT12\FORTRAN\SAMPLES\PM\DLG2FOR
```

In addition to the source code, each directory also contains the resource and include files used by the samples.

This chapter and the chapter that follows it walk you through the code of the DLG1COB and DLG2COB COBOL sample applications. The applications demonstrate window and dialog handling in a nonreentrant programming language. The chapters do not walk you through the FORTRAN samples. The English-like nature of COBOL allows the code explanations to be readily understood by FORTRAN programmers. The COBOL samples use the same calls as the FORTRAN samples, the variables have the same names, and the contents of the files used by the program are the same, except where indicated otherwise. The files used by the FORTRAN samples have the file names DLG1FOR and DLG2FOR. Any non-obvious differences between the COBOL and FORTRAN samples are described.

The DLG1COB.COB Dialog Box Sample Application

The DLG1COB sample program uses a dialog box to obtain data from the user. The program uses a standard window to display its main window. The standard window has an action bar with the menu items **Commands** and **Exit**. The client area has a blank white background.

When the user selects the **Commands** item in the action bar, a pull-down appears, containing the item **Where...**, as shown in Figure 11-1.

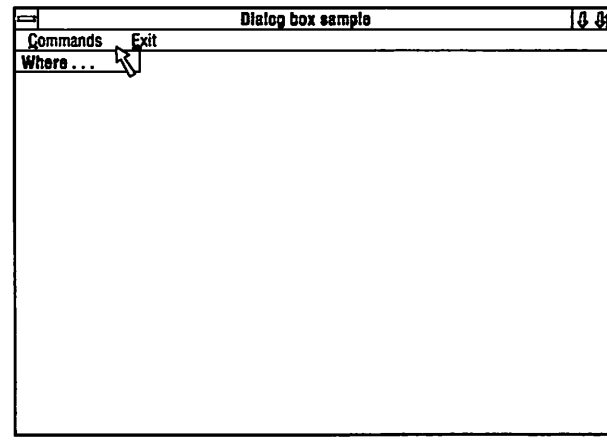


Figure 11-1. DLG1COB Sample Application Window

The ellipsis (...) indicates to the user that a pop-up window follows from the item. When the user selects **Where...**, the program responds by displaying a pop-up window, implemented in the program by a dialog box.

The dialog box contains the text **Where do you live?**, and a text entry field that is blank. The dialog box also contains two pushbuttons, one with the text **Enter**, the other with the text **Cancel**. Selecting **Cancel** causes the dialog box to disappear without any other effect.

Text can be entered into the entry field, for example, **Winchester** as shown in Figure 11-2.

When the user selects the **Enter** pushbutton, the dialog box disappears and the name of the town or city appears in the window, at the end of a string of text; for example:

You live in
Winchester

The **Exit Program** option on the second pull-down menu causes the program to terminate.

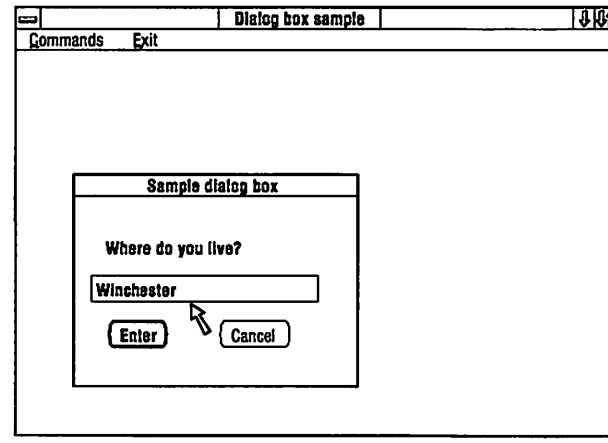


Figure 11-2. DLG1COB Sample Application Window with Dialog Box

Example Code: Dialog Resource Script File

The resource script file contains the definitions of the following resources:

- Icon, used to represent the main window when it is minimized
- Accelerator table
- Action bar and pull-down
- Text string.

The code for these definitions follows:

```
#include <os2.h>
#include "DLG1COB.h"

ICON       ID_MAINWND PRELOAD DLG1COB.ico

ACCELTABLE ID_MAINWND PRELOAD
BEGIN
  VK_F3, ID_EXITPROGRAM, VIRTUALKEY
END

MENU       ID_MAINWND PRELOAD
BEGIN
  SUBMENU "~Commands", ID_COMMANDS
  BEGIN
    MENUITEM "~Where...",        ID_WHERE,       MIS_TEXT
  END
  SUBMENU "E~xit", ID_EXIT
  BEGIN
    MENUITEM "E~xit Dlg1Cob\tF3", ID_EXITPROGRAM, MIS_TEXT
    MENUITEM "~Resume",           ID_RESUME,      MIS_TEXT
  END
END

STRINGTABLE PRELOAD
BEGIN
```

```
  ID_RESPONSE, "You live in "
END

rcinclude DLG1COB.dlg
```

In the resource script file, the rcinclude statement points to the file DLG1COB.DLG. The file contains the template of a dialog box created by the dialog box editor. (Chapter 7, "Dialog Boxes and Controls" tells you how to specify dialog boxes and their contents.) The DLG1COB.DLG file is compiled into the DLG1COB.RES file along with the other resources used by the program module.

Example Code: Dialog Template File

Example code from DLG1COB

```
DLGINCLUDE 1 "DLG1COB.H"

DLGTEMPLATE ID_MYDIALOG PRELOAD MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Sample dialog box", ID_MYDIALOG, 91, 90, 111, 59,
           WS_VISIBLE | WS_SAVEBITS,
           FCF_DLGBORDER | FCF_NOBYTEALIGN | FCF_TITLEBAR
    BEGIN
        CONTROL "Where do you live?", ID_NULL, 9, 43, 91, 10, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "", ID_ENTRYFIELD, 10, 27, 87, 8, WC_ENTRYFIELD,
                ES_LEFT | ES_AUTOSCROLL | ES_MARGIN |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Enter", 1, 6, 4, 40, 12, WC_BUTTON,
                BS_PUSHBUTTON | BS_DEFAULT | WS_GROUP | WS_TABSTOP |
                WS_VISIBLE
        CONTROL "Cancel", 2, 60, 4, 40, 12, WC_BUTTON,
                BS_PUSHBUTTON | WS_VISIBLE
    END
END
```

Example Code: Dialog Include File

The DLG1COB.H file equates the symbolic identifiers, used by the resource compiler to identify the resources, to their integer values:

```
#define ID_MAINWND      256
#define ID_MYDIALOG     257
#define ID_ENTRYFIELD   258
#define ID_COMMANDS     259
#define ID_WHERE        260
#define ID_EXIT         261
#define ID_EXITPROGRAM  262
#define ID_RESUME       263
```

```
#define ID_RESPONSE     264
#define ID_NULL         9999
```

You must define the COBOL or FORTRAN equivalents of the resource identifiers in an include file for the program. For example, the program DLG1COB.CBL uses the following identifiers, which are defined in the file DLG1COB.CIN:

```
77  ID-MAINWIND pic s9(9) usage is binary value 256.
77  ID-MYDIALOG pic s9(9) usage is binary value 257.
77  ID-ENTRYFIELD pic s9(9) usage is binary value 258.
77  ID-WHERE pic s9(9) usage is binary value 260.
77  ID-EXITPROGRAM pic s9(9) usage is binary value 262.
77  ID-RESPONSE pic s9(9) usage is binary value 264.
77  ID-NULL pic s9(9) usage is binary value 9999.
```

The program DLG1FOR.FOR uses identifiers defined in the file DLG1FOR.FIN:

```
 INTEGER*4 ID_MAINWIND, ID_MYDIALOG, ID_ENTRYFIELD,
1          ID_WHERE, ID_EXITPROGRAM, ID_RESPONSE,
2          ID_NULL
 PARAMETER ( ID_MAINWIND = 256)
 PARAMETER ( ID_MYDIALOG = 257)
 PARAMETER ( ID_ENTRYFIELD = 258)
 PARAMETER ( ID_WHERE = 260)
 PARAMETER ( ID_EXITPROGRAM = 262)
 PARAMETER ( ID_RESPONSE = 264)
 PARAMETER ( ID_NULL = 9999)
```

COBOL Program Structure

In COBOL, all records, files, constants, and variables used by a program are contained in the data division of the source code. The working storage section of the data division contains data areas that are developed and processed internally by the program. Separate from the data division, the code of procedures and routines that form the solution to the programming problem are contained in or called from the procedure division.

Example Code: DLG1COB Data Division

```
 data division.
 working-storage section.

 COPY 'DLG1COB.CIN'.

*** Global variables ***

 77  LENGTH-STRING pic s9(9) usage is binary value 32.
 77  hab pic s9(9) usage is binary.
```

```
 77  errorcode pic s9(9) usage is binary.
 77  hmq pic s9(9) usage is binary.
 77  hwndClient pic s9(9) usage is binary.
 77  flCreate pic s9(9) usage is binary.
 77  bComplete pic s9(9) usage is binary.
 77  hwndFrame pic s9(9) usage is binary.
 77  szLocation pic x(33).
 77  szLocationLen pic s9(9) usage is binary value 0.
 77  szResponse pic x(33).
 77  szResponseLen pic s9(9) usage is binary value 0.
 77  hwndDialg pic s9(9) usage is binary.
 77  dummy pic s9(9) usage is binary.

*** End of global variable declarations ***

*** local variables for main procedure ***

 01  qmsg.
 COPY   '..\..\..\INCLUDE\QMSG.CIN'.
 77  szClassName pic x(8) value "MyWindow".
 77  szClassNameLen pic s9(9) usage is binary value 8.
 77  szWinTitle pic x.
 77  szWinTitleLen pic s9(9) usage is binary value 0.
 77  quitVal pic s9(9) usage is binary.
 77  queueSize pic s9(9) usage is binary value 0.
 77  extra pic s9(9) usage is binary value 0.
 77  frameStyle pic s9(9) usage is binary.
 01  Setbitsdata.
     02 SetbitsdataArray occurs 32 times.
         03  SetbitsdataArrayElmt pic s9(9) usage is binary.
 77  styleClient pic s9(9) usage is binary value 0.
 77  firstMsg pic s9(9) usage is binary value 0.
 77  lastMsg pic s9(9) usage is binary value 0.

*** Local variables for MyWindowRoutine ***

 77  hps pic s9(9) usage is binary.
 77  ptx pic s9(9) usage is binary.
 77  pty pic s9(9) usage is binary.
 77  rcxleft pic s9(9) usage is binary.
 77  rcybottom pic s9(9) usage is binary.
 77  rcxright pic s9(9) usage is binary.
 77  rcytop pic s9(9) usage is binary.
 77  winMp1Len pic s9(9) usage is binary value 2.
 01  winMp1Types.
     02 winMp1TypesArray occurs 2 times.
         03  winMp1TypesArrayElmt pic s9(9) usage is binary.
 01  winMp1Vals.
     02 winMp1ValsArray occurs 2 times.
         03  winMp1ValsArrayElmt pic s9(9) usage is binary.
 77  winCommandPosition pic s9(9) usage is binary value 2.
 77  winShort1 pic s9(9) usage is binary.
 77  mparamVal pic s9(9) usage is binary value 0.

*** Local variables for MyDlgRoutine ***

 77  dlgMp1Len pic s9(9) usage is binary value 2.
 01  dlgMp1Types.
     02 dlgMp1TypesArray occurs 2 times.
        03  dlgMp1TypesArrayElmt pic s9(9) usage is binary.
```

```
01  dlgMp1Vals.
    02 dlgMp1ValsArray occurs 2 times.
       03  dlgMp1ValsArrayElmt pic s9(9) usage is binary.

77  dlgCommandPosition pic s9(9) usage is binary value 2.
77  dlgShort1 pic s9(9) usage is binary.
77  hwnd pic s9(9) usage is binary.

*** End of data declarations ***
```

DLG1COB Data Division Walk Through

Including Constants

The first statement in the working storage section is:

```
COPY 'DLG1COB.CIN'.
```

The statement copies the file DLG1COB.CIN into the source. The file contains definitions of constants used by PM, copied from the PM include file PM.CIN. DLG1COB.CIN also contains constants used by the sample program.

The FORTRAN version of the sample, DLG1FOR, contains a statement that copies in the equivalent file for FORTRAN:

```
INCLUDE 'DLG1FOR.FIN'
```

Another approach is to copy in the PM constants file, and copy in a separate file for your own program constants.

Variables and Structures

The remainder of the working storage section of DLG1COB defines global variables, and variables that are local to each routine in the procedure division. (The FORTRAN sample copies its global variables from a file DLG1FORC.FIN. Local variables are declared in the routines as required.)

A particular point of interest in the local variables for the main procedure are the statements that define the qmsg structure used in the main procedure:

```
01  qmsg.
COPY   '..\..\..\INCLUDE\QMSG.CIN'.
```

In COBOL, a program defines each data structure in a hierarchical form, where 01 is the highest level in the hierarchy. The program defines the 01 level, and the COPY statement copies in the file QMSG.CIN, which contains the hierarchical levels subordinate to the 01 level for the qmsg structure. In its message-processing loop, the program reads messages from the input queue into the qmsg structure:

```
    10 QMSG-BLOCK.
***   Window handle
        15 QMSG-HWND    PIC S9(9) USAGE BINARY.
***   Message identity
        15 QMSG-MSGID   PIC S9(9) USAGE BINARY.
***   Parameter 1
        15 QMSG-PARAM1  PIC S9(9) USAGE BINARY.
***   Parameter 2
        15 QMSG-PARAM2  PIC S9(9) USAGE BINARY.
***   Message time
        15 QMSG-TIME    PIC S9(9) USAGE BINARY.
***   Pointer position when message was generated
        15 QMSG-POINT.
          20 QMSG-X     PIC S9(9) USAGE BINARY.
          20 QMSG-Y     PIC S9(9) USAGE BINARY.
```

The 01 level is not defined in the include file, so you can create more than one instance of a structure by defining your own 01 levels with different names, and copying in the structure file after each 01 level.

For FORTRAN, the file QMSG.FIN contains the entire qmsg structure defined as a character array. Each component of the structure is defined separately as an integer. EQUIVALENCE statements redefine each component as an element of the character array:

```
C
C**** Constant for length of block (bytes)
      INTEGER*4            QMSG_LEN
      PARAMETER (QMSG_LEN=28)
C
C**** Basic block as CHAR string (to copy to/from)
      CHARACTER*(QMSG_LEN) QMSG_BLOCK
C
C**** DECLAREs for structure components
C**** Window handle
      INTEGER*4            QMSG_HWND
C**** Message identity
      INTEGER*4            QMSG_MSGID
C**** Parameter 1
      INTEGER*4            QMSG_PARAM1
C**** Parameter 2
      INTEGER*4            QMSG_PARAM2
C**** Message time
      INTEGER*4            QMSG_TIME
C**** Pointer position when message was generated
      INTEGER*4            QMSG_X
      INTEGER*4            QMSG_Y
C
C**** EQUIVALENCEs for structure components
      EQUIVALENCE(QMSG_HWND  ,QMSG_BLOCK( 1: 4))
      EQUIVALENCE(QMSG_MSGID ,QMSG_BLOCK( 5: 8))
      EQUIVALENCE(QMSG_PARAM1,QMSG_BLOCK( 9:12))
      EQUIVALENCE(QMSG_PARAM2,QMSG_BLOCK(13:16))
      EQUIVALENCE(QMSG_TIME  ,QMSG_BLOCK(17:20))
      EQUIVALENCE(QMSG_X     ,QMSG_BLOCK(21:24))
      EQUIVALENCE(QMSG_Y     ,QMSG_BLOCK(25:28))
```

To include the structure in your program source, use the following statement:

```
INCLUDE '..\..\..\INCLUDE\QMSG.FIN'
```

In your program, you use the basic block QMSG_BLOCK as the character string parameter on the calls that use the structure.

You can declare more than one instance of the structure. For example, the following statement declares two character strings MY_QMSG1 and MY_QMSG2:

```
CHARACTER*(QMSG_LEN) MY_QMSG1 MY_QMSG2
```

Other data declarations to note in the sample program are Setbitsdata, winMp1Types, winMp1Vals, dlgMp1Types, and dlgMp1Vals. Setbitsdata is a structure holding a 32-element array of signed integers, and is used by the call WUSBIT (WinSetBits) in the main procedure. winMp1Types, winMp1Vals, dlgMp1Types, and dlgMp1Vals are used by the call WUQVAL (WinQueryValue) which extracts data from message parameters.

Example Code: DLG1COB Main Procedure

```
                 procedure division.

                 initialize-globals section.

                *** Initialize global data ***

                     move FCF-STANDARD to flCreate
                     move FALSE-VALUE to bComplete
                     move NULL to hwndDialg.
```

```
*******************************
*  Start of Main procedure    *
*******************************

 main-procedure section.


*** Initialize local variables ***

* Initialize elements of data array for setting framestyle flag
* to WS-VISIBLE (top bit set cannot be assigned directly)

     initialize Setbitsdata replacing numeric by 0
     move 1 to SetbitsdataArrayElmt(1)

*** end of initialization ***

     call "WMINIT" using NULL, hab

* create message queue

     call "WMCMQ" using hab, queueSize, hmq

* Load "You live in" string from resource

     call "WMLSTR" using
                         hab,
                         NULL,
                         ID-RESPONSE,
                         LENGTH-STRING,
                         szResponse,
                         szResponseLen

* register window class

     call "WMRCL" using
                        hab,
                        szClassNameLen,
                        szClassName,
                        NULL,
                        CS-SIZEREDRAW,
                        extra,
                        dummy

* Set framestyle to WS-VISIBLE using setbits

     call "WUSBIT" using
                          hab,
                          framestyle,
                          Setbitsdata,
                          dummy

* Create standard window

     call "WICRTS" using
                         HWND-DESKTOP,
                         frameStyle,
                         flCreate,
                         szClassNameLen,
```

```
                        szClassName,
                        szWinTitleLen,
                        szWinTitle,
                        styleClient,
                        NULL,
                        ID-MAINWIND,
                        hwndClient,
                        hwndFrame.

*****************************************************************
* Get and dispatch messages from the application message queue *
* until WMGET returns FALSE, indicating a WM-QUIT message.     *
* Execute window and dialog procedures that have messages.     *
*****************************************************************

 process-message.

     call "WMGET" using
                     hab,
                     qmsg,
                     NULL,
                     firstMsg,
                     lastMsg,
                     quitVal

     if quitVal = FALSE-VALUE GO TO closedown.

* If handle in message is the handle of my window then execute
* the window routine, else dispatch message to the system

     if QMSG-HWND = hwndClient

*       Call MyWindowRoutine to handle panel of window

        perform MyWindowRoutine through EndWindowRoutine

     else if QMSG-HWND = hwndFrame

             if QMSG-MSGID = WM-COMMAND
*               Call MyWindowRoutine to handle the commands
                perform MyWindowRoutine through EndWindowRoutine
              else
*               Dispatch message to frame procedure
                call "WMDISP" using hab, qmsg, dummy

     else if QMSG-HWND = hwndDialg

* Call MyDlgRoutine to handle dialog box

         perform MyDlgRoutine through EndDlgRoutine

     else

* Otherwise dispatch message

         call "WMDISP" using hab, qmsg ,dummy.

* Free Message

     call "WUFMSG" using hab,
```

```
                                QMSG-HWND,
                                QMSG-MSGID,
                                QMSG-PARAM1,
                                QMSG-PARAM2,
                                NULL,
                                dummy

     go to process-message.

* Tidy up

 closedown.

* Free Message

     call "WUFMSG" using hab,
                         QMSG-HWND,
                         QMSG-MSGID,
                         QMSG-PARAM1,
                         QMSG-PARAM2,
                         NULL,
                         dummy

     call "WIDEL" using hwndFrame, dummy
     call "WMDMQ" using hmq, dummy

* and terminate the application

     call "WMTERM" using hab, dummy
     stop run.

*****************************************************************
*                   End of main procedure                       *
*****************************************************************
```

DLG1COB Main Procedure Walk Through

Initialization

The main procedure starts by initializing three variables with their PM constant values from the include files. (The constants are in uppercase.) The procedure initializes the Setbitsdata array, and places a 1 in its first element. Setbitsdata is used by the WUSBIT (WinSetBits) call later in the procedure.

The WMINIT (WinInitialize) call initializes the PM application programming environment. WMINIT outputs hab, the handle of an anchor block, which is an area of PM-internal resources allocated to the program. The anchor block handle is used as an input parameter to many calls.

The WMCMQ (WinCreateMsgQueue) call creates the program's message input queue. The call outputs hmq, the message queue handle, also used on some subsequent calls.

The WMLSTR (WinLoadString) call loads the string "You live in " from ID_RESPONSE in the resource file into szResponse.

The WMRCL (WinRegisterClass) call registers a window class with a NULL window procedure. szClassName contains the class name "MyWindow". Any message that is sent to a window of this class is intercepted by PM's language-support window procedure. The CS_SIZEREDRAW class style specifies that the contents of the window are resized when the window boundaries are resized. See the *Programming Reference: Volume 2* for more details.

Initializing PM 4-Byte Integers

The 8-bit, 16-bit, and 32-bit strings that the Presentation Manager uses all map to the COBOL picture description:

```
PIC S9(9) USAGE BINARY
```

and INTEGER*4 in FORTRAN. In each case, the bit string resides in the least significant bytes in the integer. A bit string can contain one or bit fields, with each bit field consisting of one or more bits, the settings of which are significant to PM. PM provides named constants for each valid *value* of each bit field.

Normally, you do not have to know how the constants are held, or what the bit settings are. For all 8- and 16-bit constants, and for most of the 32-bit constants, it is sufficient for you to know the name of the constant value, and to assign or add its value to the call parameter. For example, the call WMRCL (WinRegisterClass) in the sample program uses the constant CS-SIZEREDRAW as its fifth parameter. As a further example, at the start of the main procedure, the constant value FCF-STANDARD is assigned to flCreate, which the call WICRTS (WinCreateStdWindow) subsequently uses as its third parameter. If a combination of like constants are needed, they can be added together, because the bit field for each constant value is held in a different position in the bit string. For example, several FCF- values can be added to result in flCreate, before being passed as the call parameter.

Some 32-bit constants have to be handled differently, however. A 32-bit constant that has either or both of its two most significant bits set to 1 has a decimal value of 10 digits. For example, the window-style constant WS-VISIBLE has the following decimal value:

```
2 147 483 648
```

Also, if a 32-bit constant that has its third most significant bit set to 1 is added to a smaller number, it can result in a 10-digit number. The constants that are or can result in 10-digit numbers in COBOL are:

WS-VISIBLE

WS-DISABLED

WS-CLIPCHILDREN

CS-CLIPCHILDREN

FCF-DBE-APPSTAT

ETYPE-GEBB

ETYPE-GEPEL.

In COBOL, a 32-bit binary number is expressed as the picture description: PIC S9(9) USAGE IS BINARY. Even though such a field is 32 bits, the largest number that the COBOL/2 compiler lets your COBOL/2 code place in it is a 9-digit number. Therefore, you have to use the WUSBIT (WinSetBits) call to do it for you.

In FORTRAN, only the most significant bit is a problem. The constants that result in 10-digit numbers in FORTRAN are:

WS_VISIBLE

FCF_DBE_APPSTAT.

In both cases, to prevent programming errors, the above constants are not defined in the PM include files. Therefore, you must use the WUSBIT (WinSetBits) call. WUSBIT (WinSetBits) sets the bits in a 4-byte (32-bit) integer, according to values you specify in a 32-element integer array. A value of 0 in any of the thirty-two elements sets the corresponding bit to 0. A value greater than 0 in an element sets the corresponding bit to 1. A value less than 0 in an element has no effect on the corresponding bit. If you are adding constants together to produce a composite constant, you must do it **before** using the WUSBIT (WinSetBits) call. The 4-byte integer must not participate in any numeric operations between being set by WUSBIT (WinSetBits) and being used as a call parameter.

For querying the bits in a 4-byte integer, there is a query call WUQBIT (WinQueryBits). Calls for modifying and querying 4-byte integers are described in "Modifying and Querying PM 4-Byte Integers" on page 11-19.

Window Creation

The call WICRTS (WinCreateStdWindow) creates a standard window. The second parameter is the framestyle variable set by WUSBIT (WinSetBits), and holds the WS-VISIBLE constant. The window created belongs to the class szClassName registered by WMRCL (WinRegisterClass) earlier in the procedure. The call returns hwndClient, the handle of

the client window, and hwndFrame, the handle of the surrounding frame.

The Message-Processing Loop

The call WMGET (WinGetMsg) creates a qmsg structure in storage, gets a message from the message queue, and places it in the qmsg structure. If the message is a WM-QUIT message, the value returned in quitVal is FALSE, and the program goes to its termination routine closedown. All other messages result in a return of TRUE.

QMSG-HWND is the part of the qmsg structure that holds the handle of the window to which the message is addressed. If QMSG-HWND contains the handle of the client window, the program calls the client window routine.

When the user selects **Where...** from the **Commands** pull-down, the menu control posts a WM-COMMAND message to the frame window. The frame sends the WM-COMMAND to the application window. Because it is sent, the application window does not receive it, so the program monitors the messages that are posted to the frame. If QMSG-HWND contains the handle of the frame window, the program calls the client window routine, which processes the WM-COMMAND message, before using WMDISP (WinDispatchMsg) to dispatch the message to the system-provided frame window procedure. If QMSG-HWND contains the handle of the program's dialog box, the program calls its dialog routine. If QMSG-HWND contains the handle of any other window, it is a system-provided window, so the program calls WMDISP (WinDispatchMsg) to dispatch the message to the window.

When the program has dealt with the message, it **must** release the storage that PM allocates internally for the qmsg structure. This is achieved by the WUFMSG (WinFreeMsg) call. This is a requirement for COBOL and FORTRAN only.

Application Termination

The termination routine calls WUFMSG (WinFreeMsg) to destroy any remaining message-associated storage, calls WIDEL (WinDestroyWindow) to destroy the program's main window, calls WMDMQ (WinDestroyMsgQueue) to destroy the message queue, and WMTERM (WinTerminate) to terminate the program's use of PM.

Example Code: DLG1COB Client Window Routine

```
*******************************
*  Start of Window Routine    *
*******************************

 MyWindowRoutine section.

*** Initialize local variables *

     move DTYP-SHORT to winMp1TypesArrayElmt(1)
     move DTYP-SHORT to winMp1TypesArrayElmt(2)

*** End of initialization ***

* Routine for main window.
* Does various actions according to messages passed to it.

     if QMSG-MSGID = WM-COMMAND

*****************************************************************
* For the Commands pull-down, when the user chooses "Where...",   *
* the application displays a dialog box.                          *
* For the Exit pull-down, the application posts itself a WM-CLOSE *
```

```
* message if Exit is chosen.                                          *
**********************************************************************

* Extract the command value from the parameter structure

            call "WUQVAL" using
                                hab,
                                QMSG-PARAM1,
                                winMp1Len,
                                winMp1Types,
                                winMp1Vals,
                                dummy

         move winMp1ValsArrayElmt(2) to winShort1

         if winShort1 = ID-WHERE

             call "WCLD" using
                              HWND-DESKTOP,
                              hwndFrame,
                              NULL,
                              NULL,
                              ID-MYDIALOG,
                              NULL,
                              hwndDialg

         else if winShort1 = ID-EXITPROGRAM

             call "WMPOST" using
                              QMSG-HWND,
                              WM-CLOSE,
                              mparamVal,
                              mparamVal,
                              dummy
         else

             call "WMDWP" using
                              QMSG-HWND,
                              QMSG-MSGID,
                              QMSG-PARAM1,
                              QMSG-PARAM2,
                              dummy

     else if (QMSG-MSGID = WM-PAINT) OR (QMSG-MSGID = WM-PPAINT)

**********************************************************************
* Window contents are drawn here. First time through, bComplete      *
* is FALSE, so window is simply filled with SYSCLR-WINDOW.           *
* On subsequent passes, if bComplete has been set to TRUE in the     *
* dialog procedure, GICHAR draws the text.                           *
**********************************************************************

* Begin Paint

         call "WIBPNT" using
                             QMSG-HWND,
                             NULL,
                             rcxleft,
                             rcybottom,
                             rcxright,
```

```
                                rcytop,
                                hps

* Fill Rectangle

        call "WIFRCT" using
                                hps,
                                rcxleft,
                                rcybottom,
                                rcxright,
                                rcytop,
                                SYSCLR-WINDOW,
                                dummy

        if bComplete = TRUE-VALUE

* Text color same as PM

            call "GSCOL" using hps, CLR-NEUTRAL

* Coordinates of "you live in"

            move 50 to ptx
            move 50 to pty

* Draw character string

            call "GICHAR" using
                                hps,
                                ptx,
                                pty,
                                szResponseLen,
                                szResponse

* Coordinates of location entered by user

            move 50 to ptx
            move 30 to pty

* Draw character string

            call "GICHAR" using
                                hps,
                                ptx,
                                pty,
                                szLocationLen,
                                szLocation
        end-if

* Drawing is complete

        call "WIEPNT" using hps, dummy

    else if QMSG-MSGID = WM-CLOSE

* Post termination message

        call "WMPOST" using
                            QMSG-HWND,
                            WM-QUIT,
```

```
                                mparamVal,
                                mparamVal,
                                dummy

           else

       * Must have WMDWP in your window procedure to process other messages

               call "WMDWP" using
                                QMSG-HWND,
                                QMSG-MSGID,
                                QMSG-PARAM1,
                                QMSG-PARAM2,
                                dummy.
        EndWindowRoutine.

      ******************************************************************
      *                    End of Window Routine                       *
      ******************************************************************
```

DLG1COB Client Window Routine Walk Through

The window routine MyWindowRoutine explicitly processes the WM-COMMAND, WM-PPAINT, and WM-CLOSE messages, and passes all others to the default window procedure.

MyWindowRoutine initializes the two elements of the winMp1Types array with the constant DTYP-SHORT. These are used by the WUQVAL (WinQueryValue) call in the WM-COMMAND processing below.

The WM-COMMAND Message

When the user selects the **Where...** choice from the **Commands** pull-down, a WM-COMMAND message with a command value of ID-WHERE is posted to the program. When the user selects the **Exit Dlg1Cob** choice from the **Exit** pull-down, a WM-COMMAND message with a command value of ID-EXITPROGRAM is posted to the program. The least significant half of the first parameter of the message contains the command value.

Constructing and Decomposing PM 4-byte Integers

The WUQVAL (WinQueryValue) call decomposes a 4-byte integer value into its component 1-byte or 2-byte integer values, or bit values. The DLG1COB sample program uses it to extract the component parts of message parameters. The component parts of each message are described in the *Programming Reference: Volume 2*. The call can be used to decompose a field of any composition.

In the sample program, the second parameter of WUQVAL is QMSG-PARAM1, the first message parameter of WM-COMMAND in the qmsg structure. The third parameter of WUQVAL specifies the number of components to be extracted from the message parameter. The example extracts two components. Message parameter components vary in their data types. The fourth parameter of WUQVAL is an array that lets you specify the data type of each component you are extracting. The example uses a two-element array winMp1Types, as it is extracting two values only. In both the elements of the array, the constant DTYP-SHORT specifies that both components are 2-byte signed integers.

The complete list of data types is:

DTYP_BIT8	8-bit value
DTYP-BIT16	16-bit value
DTYP-BYTE	1-byte signed integer
DTYP-UCHAR	1-byte unsigned integer
DTYP-SHORT	2-byte signed integer
DTYP-USHORT	2-byte unsigned integer.

WUQVAL (WinQueryValue) returns the data values of the extracted components in an array in the fifth parameter. The command value is moved to winShort1.

There is also a call WUSVAL (WinSetValue) for composing 4-byte integers from component 1-byte integers, 2-byte integers, and bit values. There is an example of the use of WUSVAL to construct a message in "The MI-CHECKBOX Command Value" on page 12-12.

The ID-WHERE Command Value

If winShort1 is ID-WHERE, the program calls WCLD (WinLoadDlg) to create the dialog box. The call specifies a NULL dialog procedure in its third parameter, which ensures that the language-support dialog procedure deals with sent messages. (The dialog box is modeless, so for messages that are posted to the dialog box the program calls the dialog routine from the message-processing loop.) The dialog box is created from the resource ID-MYDIALOG in the dialog template.

The ID-EXITPROGRAM Command Value

If winShort1 is ID-EXITPROGRAM, the program posts itself a WM-CLOSE message, which it processes later in the window routine.

If the command value of WM-COMMAND is neither ID-WHERE or ID-EXITPROGRAM, the default window procedure WMDWP (WinDefWindowProc) is called to deal with it.

The WM-PPAINT Message

When the program's main window is first created, a WM-PAINT message is sent to the program, and subsequently posted on to the program by the language-support window procedure as a WM-PPAINT message.

When the dialog routine has some text to display, it calls a function that sends a WM-PAINT message, which is posted on to the program by the language-support window procedure as a WM-PPAINT. However, the window routine also tests for the WM-PAINT message in case this is posted straight to the program.

Painting is started by calling WIBPNT (WinBeginPaint). The call specifies NULL, to get a cache micro presentation space, specifies the coordinates of the update region, and returns the handle of the presentation space.

The call WIFRCT (WinFillRect) fills the window with the color SYSCLR-WINDOW.

The first time through the WM-PAINT processing, the switch bComplete is equal to FALSE-VALUE, as initialized by the program, so the processing drops through to the end of the IF.

When the dialog routine has some text for the window routine to display, it sets the switch bComplete to TRUE-VALUE.

The call GSCOL (GpiSetColor) sets the color of the text to CLR-NEUTRAL.

GICHAR (GpiCharStringAt) writes the string "You live in", held in szResponse, at the ptx and pty coordinates 50 and 50. GICHAR (GpiCharStringAt) subsequently writes the string szLocation, obtained by the dialog routine, at the ptx and pty coordinates 50 and 30.

The call WIEPNT (WinEndPaint) ends the processing of the WM-PAINT message.

The WM-CLOSE Message

In the WM-COMMAND processing, the program posts itself a WM-CLOSE message when the user selects the **Exit Dlg1Cob** choice from the **Exit** pull-down. A WM-CLOSE is also posted when the user closes the main window from the system menu.

The program processes the WM-CLOSE message by posting itself a WM-QUIT message. In the message-processing loop, when the call WMGET (WinGetMsg) finds a WM-QUIT message on the queue, the call returns a value of FALSE-VALUE, which causes the program to go to its close-down routine.

Default Window Processing

For any posted messages that it does not explicitly process, the window routine calls WMDWP (WinDefWindowProc), the default window procedure.

Example Code: DLG1COB Dialog Routine

```
*******************************
*  Start of dialog procedure  *
*******************************

 MyDlgRoutine section.

*** Initialize local variables ***

     move DTYP-SHORT to dlgMp1TypesArrayElmt(1)
     move DTYP-SHORT to dlgMp1TypesArrayElmt(2)

*** End of initialization ***

* Procedure for dialog box.
* Reads a string into szLocation

* switch on QMSG-MSGID

     if QMSG-MSGID = WM-COMMAND

*********************************************************************
* PM sends a WM-COMMAND message when the user presses either      *
* the Enter or Escape pushbuttons.                                  *
*********************************************************************

* Extract the command value from the parameter structure

         call "WUQVAL" using
                          hab,
                          QMSG-PARAM1,
                          dlgMp1Len,
                          dlgMp1Types,
                          dlgMp1Vals,
                          dummy

         move dlgMp1ValsArrayElmt(2) to dlgShort1

* The Enter pushbutton or key

         if dlgShort1 = DID-OK

* WIQTXT copies the contents of the entry field into szLocation

             call "WIQHID" using
                           QMSG-HWND,
                           ID-ENTRYFIELD,
                           hwnd
             call "WIQTXT" using
                           hwnd,
                           LENGTH-STRING,
```

```
                                szLocation,
                                szLocationLen

* Set switch so strings are drawn in WM-PAINT processing.

            move TRUE-VALUE to bComplete

* Terminate dialog box

            call "WIDEL" using QMSG-HWND, dummy

* Force a repaint

            call "WINVRG" using
                                hwndClient,
                                NULL,
                                FALSE-VALUE,
                                dummy

* Case of Cancel pushbutton or Escape key

        else if dlgShort1 = DID-CANCEL

* Terminate dialog box

            call "WIDEL" using QMSG-HWND, dummy

* Force a repaint

            call "WINVRG" using
                                hwndClient,
                                NULL,
                                FALSE-VALUE,
                                dummy

    else

*****************************************************************
* Any event messages that the dialog procedure has not processed *
* come here and are processed by WCCDP.                          *
* This call MUST exist in your dialog procedure.                 *
*****************************************************************

        call "WCDDP" using
                            QMSG-HWND,
                            QMSG-MSGID,
                            QMSG-PARAM1,
                            QMSG-PARAM2,
                            dummy.

 EndDlgRoutine.

*****************************************************************
*                  End of Dialog procedure                      *
*****************************************************************
```

DLG1COB Dialog Routine Walkthrough

The dialog routine MyDlgRoutine processes only the WM-COMMAND message, and passes all others to the default dialog procedure.

MyDlgRoutine initializes the two elements of the dlgMp1Types array with the constant DTYP-SHORT. These are used by the WUQVAL (WinQueryValue) call in the WM-COMMAND processing below.

The WM-COMMAND Message

When the dialog box is displayed, the user is expected to enter some text, and select the **Enter** pushbutton. A **Cancel** pushbutton is also available.

When the user selects one of the pushbuttons, it posts a WM-COMMAND message to its owner, the dialog box. The message contains a command value of DID-OK for the **Enter** pushbutton, and a command value of DID-CANCEL for the **Cancel** pushbutton.

The call WUQVAL (WinQueryValue) extracts the command value from the message, and places it in the second element of the dlgMp1Vals array. The program puts the command value in dlgShort1, and tests the value.

DID-OK Command Value Processing

If the command value is DID-OK, the program calls WIQHID (WinWindowFromID) to get the window handle of the window with the identifier ID-ENTRYFIELD. The handle hwnd is used by the call WIQTXT (WinQueryText) to get the text from the entry field and put it in szLocation.

The bComplete switch is set to TRUE-VALUE so that the window routine is notified of text to be printed.

The call WIDEL (WinDestroyWindow) destroys the dialog box and its controls.

The call WINVRG (WinInvalidateRegion) sends a WM-PAINT message to the client window.

DID-CANCEL Command Value Processing

If the command value is DID-CANCEL, the program call performs no processing other than calling WIDEL (WinDestroyWindow) and WINVRG (WinInvalidateRegion).

Default Dialog Processing

For all messages other than WM-COMMAND, the dialog routine calls WCDDP (WinDefDlgProc), the default dialog procedure.

Modifying and Querying PM 4-Byte Integers

"Initializing PM 4-Byte Integers" on page 11-11 describes how to initialize 4-byte integer variables with constant values or combinations of constant values. This section describes modifying and querying the bits in a 4-byte integer.

For example, the GIPCRT (GpiCreatePS) call creates a presentation space. Its fourth parameter is a 4-byte integer input parameter that lets you set various options for the presentation space. The options are passed as a bit string, consisting of the bit fields PS-UNITS (6 bits), PS-FORMAT (4 bits), PS-TYPE (1 bit), PS-MODE (1 bit), and PS-ASSOCIATE (1 bit). All but one of the bit fields can hold one of several named constants:

Bit Field	Named Constants
PS-UNITS	PU-ARBITRARY
	PU-PELS
	PU-LOMETRIC
	PU-HIMETRIC
	PU-LOENGLISH
	PU-HIENGLISH
	PU-TWIPS
PS-FORMAT	GPIF-DEFAULT
	GPIF-SHORT
	GPIF-LONG
PS-TYPE	GPIT-NORMAL
	GPIT-MICRO
PS-MODE	Reserved
PS-ASSOCIATE	GPI-NOASSOC
	GPIA-ASSOC

For example, to initialize an options variable in COBOL, you could code:

```
COMPUTE OPTIONS = PU-ARBITRARY + GPIF-SHORT
                + GPIT-MICRO + GPIA-ASSOC.
```

The equivalent statement in FORTRAN is similar.

To modify a bit field in COBOL or FORTRAN, you use the call WUSBM (WinSetBitsUnderMask). The call modifies bits in a 4-byte integer, under the control of a mask. To query a bit field, you use the call WUQBM (WinQueryBitsUnderMask), which queries bits in a 4-byte integer, under the control of a mask. There is an example of the call WUQBM in "The MI-CHECKBOX Command Value" on page 12-12.

The mask is itself a named constant that identifies the bit field whose bit strings you want to modify or query. Only the mask bits that correspond to the bit field you want to modify or query are set to 1. So, the mask literally masks off all the bit fields except the one you want to modify or query.

Here is an example of the use of the query and modify calls in COBOL:

```
* Query current setting of PS-FORMAT field
CALL "WUQBM" USING HAB, OPTIONS, PS-FORMAT,
                   VALUE, SUCCESS.
*
* Test current value of field - if default,
  set to short
IF VALUE = GPIF-DEFAULT
CALL "WUSBM" USING HAB, OPTIONS, PS-FORMAT,
                   GPIF-SHORT, SUCCESS.
```

In the above example, the call WUQBM (WinQueryBitsUnderMask) queries the 4-byte integer OPTIONS, using the PS-FORMAT mask so that only that bit field is returned in VALUE. VALUE is tested to see if it contains the value GPIF-DEFAULT. If it does, the call WUSBM (WinSetBitsUnderMask) sets the PS-FORMAT bit field of OPTIONS to the value of the named constant GPIF-SHORT.

Bit-Management Call Summary

WUSBIT (WinSetBits)
Modifies bits in a 4-byte integer under the control of an array of thirty-two integers.

WUSBM (WinSetBitsUnderMask)
Modifies bits in a 4-byte integer under the control of a bit mask.

WUQBIT (WinQueryBits)
Queries bits in a 4-byte integer, returning the bits as an array of thirty two integers.

WUQBM (WinQueryBitsUnderMask)
Queries bits in a 4-byte integer under the control of a bit mask.

Chapter 12. The DLG2 Nonreentrant Sample Application

The DLG2COB.COB program demonstrates how to use many of the control windows in dialog boxes. Its main window is a standard window containing a system menu, size borders, title, and an action bar. The action bar contains the choices **Dialogs**, **Exit**, and **Help**.

The **Dialog** pull-down contains the following choices:

Entry field example
Auto radio button example
Check box example
List box example.

Each choice displays a dialog box containing the controls indicated by the title of the choice. User interaction with each dialog box affects the contents of the main window. The entry field dialog box lets the user enter integers. When the user submits the integers to the program by selecting the **Enter** push-button, the integers are displayed in the main window. The auto radio button example lets you select a color, which is first shown to the user in a preview window in the dialog box. When the user submits the color setting to the program, a color sample window in the main window is set to the chosen color. The check box dialog box operates in a similar way to the radio button dialog box, but allows the user to set a color by mixing colors.

The level of complexity of the program's user interface is close to that of a real application. Consequently, its source code is long. So this chapter does not list in full the source code and all its associated files. Instead, sections of code from the program are used to illustrate the techniques involved in programming dialog boxes and controls. You can print the DLG2COB or DLG2FOR programs and associated files yourself, and use the listings in conjunction with this chapter.

The structure of DLG2COB.COB, and its FORTRAN equivalent, consists of a main routine, window routines for the program's main window, color preview window, and color sample window, dialog routines for each of the dialog boxes, and several routines that are called by the other routines.

DLG2COB Data Division Walk Through

The data division contains the declarations of the constants and variables used by the program.

A particular point of interest is that, in addition to including the qmsg structure to hold messages, the program also includes an swp structure:

```
01 swp.
COPY   '..\..\..\INCLUDE\SWP.CIN'.
```

The structure is an output parameter of the call WIQPOS (WinQueryWindowPos) in the routine CenterDlgBox that centers each dialog box on the screen. The call is an example of a call that passes a structure as a parameter. For calls that use a structure as an input or output parameter, you must include the definition of the structure in your program.

Note that, in COBOL and FORTRAN, some calls in the PM API use handles to structures, or use structures that contain handles to other structures. You deal with these structures in a particular way that the sample program illustrates, and which is summarized in "Using Data Structures" on page 12-35.

DLG2COB Main Procedure Walk Through

The main procedure initializes various data type arrays used in the program. The WMINIT (WinInitialize) and WMCMQ (WinCreateMsgQueue) calls initialize PM and create a message queue.

The variable clrSample is set to the sum of the named constants CLR-BLUE and CLR-GREEN. The sum of CLR-BLUE and CLR-GREEN is CLR-CYAN, which the program could equally well have assigned to clrSample.

A series of WMLSTR (WinLoadString) calls loads the text strings used by the program.

Three calls to WMRCL (WinRegisterClass) respectively register the program's main window, the color sample window used in the main window, and the color preview window used in the auto radio button and check box dialog boxes.

A call to WICRTS (WinCreateStdWindow) creates the main window, a call to the routine ClientCreate creates the sample window in the client area of the main window, and a message-processing loop consisting of a WMGET (WinGetMsg) and a WMDISP (WinDispatchMsg) gets messages from the queue and dispatches them to the system-provided windows. In the code between the WMGET and the WMDISP, for messages posted to any window of a class registered by the program, the window routine or dialog routine identified by the window handle is called. All other messages are dispatched to the system-provided window identified by the window handle. At the end of the message-processing loop, and in the close-down routine, the storage associated with the message structure is freed by WUFMSG (WinFreeMsg). The close down routine is the same as in the DLG1COB and DLG1FOR programs.

Example Code: MyClientWindowRoutine

```
*************************************************************************
*                    Start of First Window Routine                      *
*************************************************************************
*************************************************************************
**                                                                      *
** Window Routine : MyClientWindowRoutine                               *
**                                                                      *
** Controls the Client Area: loads various dialogs.                     *
**                                                                      *
*************************************************************************

 MyClientWindowRoutine.

* Switch on message type

* If Paint message
     if (QMSG-MSGID = WM-PAINT) OR (QMSG-MSGID = WM-PPAINT)

* Call private function ClientPaint to paint window

      perform ClientPaint through EndClientPaint

* Else If Help message

     else if (QMSG-MSGID = WM-HELP)

*************************************************************************
*                                                                       *
*  This will be received when either:-                                  *
*                                                                       *
*    1. The user presses F1                                           *
* or 2. The user clicks on the action bar item Help                     *
* or 3. The user selects a help pushbutton in a dialog                  *
*                                                                       *
*************************************************************************

* Call private Help function
```

```
       perform HelpProc through EndHelpProc

* Else If Command message

     else if (QMSG-MSGID = WM-COMMAND)

*************************************************************************
* Call Query Value to extract command value from QMSG-PARAM1.
* The required value is contained in the low word of QMSG-PARAM1.
* WUQVAL extracts elements from mp1 in high order sequence, so
* it is the second returned value that is required.
*************************************************************************

       move 2 to ValuesCount
       call "WUQVAL" using  hab,
                            QMSG-PARAM1,
                            ValuesCount,
                            mp1ValueTypes,
                            mp1ValueData,
                            dummy

       move mp1Values(2) to idCommand

* Call private function client command to process the command

       perform ClientCommand through End-ClientCommand

* Else If Close message

     else if (QMSG-MSGID = WM-CLOSE)

* Post Quit message to cause termination
       call "WMPOST" using  QMSG-HWND,
                            WM-QUIT,
                            NULL,
                            NULL,
                            dummy
     else

* Else message is passed to default window procedure
       call "WMDWP" using  QMSG-HWND,
                           QMSG-MSGID,
                           QMSG-PARAM1,
                           QMSG-PARAM2,
                           dummy.

* End of switch on message type

 EndClientWindow.

*************************************************************************
*                   End of First Window Routine                         *
*************************************************************************
```

DLG2COB Client Window Routine Walk Through

The window routine MyClientWindowRoutine explicitly processes the WM-PAINT, WM-PPAINT, WM-HELP, WM-COMMAND, and WM-CLOSE messages, and passes all others to the default window procedure.

The WM-PAINT Message

A WM-PAINT is sent by the call WINVRG (WinInvalidateRegion) in the processing of the WM-COMMAND message, which is posted a result of the user selecting a choice from a pull-down.

When a dialog box is dismissed by the user selecting the **Enter** or **Cancel** pushbuttons, the dialog routine calls WIDEL (WinDestroyWindow) to remove the dialog box from the screen, and calls WINVRG (WinInvalidateRegion) to send a WM-PAINT message to the client window.

The WM-PAINT message is posted on as a WM-PPAINT message by the language-support window procedure. The window routine processes the WM-PPAINT message by calling the routine ClientPaint to repaint the client window.

The WM-HELP Message

The WM-HELP message is processed by calling the routine HelpProc, which simply displays a message box.

The WM-CLOSE Message

The program processes the WM-CLOSE message by posting itself a WM-QUIT message. In the message-processing loop, when the call WMGET (WinGetMsg) finds a WM-QUIT message on the queue, the call returns a value of FALSE-VALUE, which causes the program to go to its close-down routine.

The WM-COMMAND Message

The text, and WM-COMMAND command value (given by the MI_ command values) for each choice in the **Dialogs** pull-down are specified in the .RC file, which is created using a text editor.

Sample Code: Dialogs pull-down in the .RC file

This example code shows the specification of the **Dialogs** pull-down, that offers a choice of four example dialog boxes.

```
SUBMENU  "~Dialogs", ID_DIALOGS
BEGIN
  MENUITEM "~Entry field example...",       MI_ENTRYFIELD,      MIS_TEXT
  MENUITEM "~Auto Radio button example...", MI_AUTORADIOBUTTON, MIS_TEXT
  MENUITEM "~Check box example...",         MI_CHECKBOX,        MIS_TEXT
  MENUITEM "~List box example...",          MI_LISTBOX,         MIS_TEXT
END
```

When the user selects an item from a pull-down, OS/2 posts a WM-COMMAND message to the program. The program calls WUQVAL (WinQueryValue) to extract the command value, and calls the routine ClientCommand to process the command value.

Example Code: ClientCommand Routine

```
*************************************************************************
**                                                                     *
**  PRIVATE FUNCTION : ClientCommand                                   *
**                                                                     *
**  Take the appropriate action when a WM-COMMAND message is received  *
**   by Client window routine.                                         *
**  Sets up the appropriate dialog template ID and pointer to the      *
**   dialog routine, for the chosen dialog type, if any.               *
**  If exit selected, post a quit message to end program.              *
**                                                                     *
*************************************************************************

 ClientCommand.

* Switch according to pull-down menu subitem chosen

* If Exit selected
     if (idCommand = MI-EXIT)

* Post Quit message to cause termination
       call "WMPOST" using  hwndWindow,
                            WM-QUIT,
                            NULL,
                            NULL,
                            dummy

* go to end of procedure
       go to End-ClientCommand

* Else If Resume selected
     else
       if (idCommand = MI-RESUME)

* go to end of procedure
         go to End-ClientCommand

* Else If Entryfield dialog selected
       else
         if (idCommand = MI-ENTRYFIELD)

           call "WCLD" using  HWND-DESKTOP,
                              hwndFrame,
                              NULL,
                              NULL,
                              DLG-ENTRYFIELD,
                              NULL,
                              hwndEntryField

* Call private function CenterDlgBox to position centrally on screen

            move hwndEntryField to hwndDlgBox
            perform CenterDlgBox through EndCenterDlgBox


* Else If Autoradiobutton dialog selected
          else
```

```
            if (idCommand = MI-AUTORADIOBUTTON)

              call "WCLD" using  HWND-DESKTOP,
                                 hwndFrame,
                                 NULL,
                                 NULL,
                                 DLG-AUTORADIOBUTTON,
                                 NULL,
                                 hwndAutoRadioButton

          * Call private function CenterDlgBox to position centrally on screen
                     move hwndAutoRadioButton to hwndDlgBox
                     perform CenterDlgBox through EndCenterDlgBox

          * Set color to Sample color
                     move clrSample to clrPreview

          * Obtain handle of Preview window
                     call "WIQHID" using  hwndAutoRadioButton,
                                          RECTL-AUTORADIOBUTTON,
                                          hwndPreview

          * Force initial paint of the Preview window in the Sample window color
                     call "WINVRG" using  hwndPreview,
                                          NULL,
                                          FALSE-VALUE,
                                          dummy

          * Else If Checkbox dialog selected

                    else
                      if (idCommand = MI-CHECKBOX)

                        call "WCLD" using  HWND-DESKTOP,
                                           hwndFrame,
                                           NULL,
                                           NULL,
                                           DLG-CHECKBOX,
                                           NULL,
                                           hwndCheckBox

          * Position dialog box centrally
                         move hwndCheckbox to hwndDlgBox
                         perform CenterDlgBox through EndCenterDlgBox

          ************************************************************************
          * Determine current color components of clrSample, and set
          * check boxes accordingly (so that Preview window is set
          * initially to same color as Sample window )
          ************************************************************************

                         move clrSample to clrPreview

          * Set Value for parameter 1 (mp1) to post to window
                         move TRUE-VALUE to mp1Values(2)
                         move 0 to mp1Values(1)

                move 2 to ValuesCount
                         call "WUSVAL" using  hab,
                                              ValuesCount,
```

```
                                        mp1ValueTypes,
                                        mp1ValueData,
                                        mp1Value,
                                        dummy

* Use Query bits under mask to extract red component bits
                call "WUQBM" using hab,
                                   clrPreview,
                                   CLR-RED,
                                   clrcomponent,
                                   dummy

* If clrPreview has a red component
                if  ( clrcomponent = CLR-RED )

* Obtain handle of red checkbox window
                  call "WIQHID" using  hwndCheckBox,
                                       CB-RED,
                                       hwndRed

* Post message to red checkbox window
                  call "WMPOST" using  hwndRed,
                                       BM-SETCHECK,
                                       mp1Value,
                                       NULL,
                                       dummy
                end-if

* Use Query bits under mask to extract green component bits
                call "WUQBM" using hab,
                                   clrPreview,
                                   CLR-GREEN,
                                   clrcomponent,
                                   dummy

* If clrPreview has a green component
                if ( clrcomponent = CLR-GREEN)

* Obtain handle of green checkbox window
                  call "WIQHID" using  hwndCheckBox,
                                       CB-GREEN,
                                       hwndGreen

* Post message to green checkbox window
                  call "WMPOST" using  hwndGreen,
                                       BM-SETCHECK,
                                       mp1Value,
                                       NULL,
                                       dummy
                end-if

* Use Query bits under mask to extract blue component bits
                call "WUQBM" using hab,
                                       clrPreview,
                                       CLR-BLUE,
                                       clrcomponent,
                                       dummy

* Else If clrPreview has a blue component
                if ( clrcomponent = CLR-BLUE )
```

```
* Obtain handle of blue checkbox window
                  call "WIQHID" using  hwndCheckBox,
                                       CB-BLUE,
                                       hwndBlue

* Post message to blue checkbox window

                  call "WMPOST" using  hwndBlue,
                                       BM-SETCHECK,
                                       mp1Value,
                                       NULL,
                                       dummy
                end-if

* Obtain handle of Preview window
               call "WIQHID" using  hwndCheckBox,
                                    RECTL-CHECKBOX,
                                    hwndPreview

* Force initial paint of Preview window in the color(s) set above
               call "WINVRG" using  hwndPreview,
                                    NULL,
                                    FALSE-VALUE,
                                    dummy

* Else If Listbox dialog selected
             else
               if (idCommand = MI-LISTBOX)

                 call "WCLD" using  HWND-DESKTOP,
                                    hwndFrame,
                                    NULL,
                                    NULL,
                                    DLG-LISTBOX,
                                    NULL,
                                    hwndListBox

* Call private function CenterDlgBox to position dialog box centrally
                 move hwndListBox to hwndDlgBox
                 perform CenterDlgBox through EndCenterDlgBox

*************************************************************************
*  Initialize the list box with a set of strings loaded from a          *
*  resource file.                                                       *
*************************************************************************

* Set value for mp1 (unchanged throughout loop)
                 move 0 to mp1Values(1)
                 move LIT-END to mp1Values(2)
                 move 2 to ValuesCount

                 call "WUSVAL" using  hab,
                                      ValuesCount,
                                      mp1ValueTypes,
                                      mp1ValueData,
                                      mp1Value,
                                      dummy

                 compute end-loop = NUMLISTBOXENTRIES - 1
```

```
                perform Initialize-Listbox through
                                EndInitialize-Listbox
                varying i from 0 by 1 until (i > end-loop)

* Set new value for second mp1
                move 0 to mp1Values(1)
                move 0 to mp1Values(2)

                call "WUSVAL" using  hab,
                                     ValuesCount,
                                     mp1ValueTypes,
                                     mp1ValueData,
                                     mp1Value,
                                     dummy

* Set value for mp2
                move 0            to mp2Values(1)
                move TRUE-VALUE to mp2Values(2)

                call "WUSVAL" using  hab,
                                     ValuesCount,
                                     mp2ValueTypes,
                                     mp2ValueData,
                                     mp2Value,
                                     dummy

* Make the first list item initially selected
                call "WCSDIM" using  hwndListBox,
                                     LISTBOX,
                                     LM-SELECTITEM,
                                     mp1Value,
                                     mp2Value,
                                     dummy

* Else go to end of procedure
              else
                go to End-ClientCommand

* End of switch on pull-down menu sub-item
              end-if
            end-if
          end-if
        end-if
      end-if
    end-if

* Force repaint of main window
    call "WINVRG" using  hwndWindow,
                         NULL,
                         FALSE-VALUE,
                         dummy.

 End-ClientCommand.
```

ClientCommand Routine Walk Through

ClientCommand has a series of IF statements that process each WM-COMMAND MI-command value that is the result of the user's selections from the pull-downs. For each user selection from the **Dialogs** pull-down, the routine calls WCLD (WinLoadDlg) to display the appropriate dialog box. At the end of the routine, a call to WINVRG (WinInvalidateRegion) repaints the client window, because the pull-down has been dismissed.

The MI-EXIT Command Value

The MI-EXIT command value occurs when the user selects **Exit Dlg2Cob** from the **Exit** pull-down. ClientCommand processes the command value by posting a WM-QUIT message to its main window.

The MI-RESUME Command Value

The MI-RESUME command value occurs when the user selects **Resume** for the **Exit** pull-down. ClientCommand processes the command value by going to the end of the routine.

The MI-ENTRYFIELD Command Value

The MI-ENTRYFIELD command value occurs when the user selects the **Entry field example...** choice from the **Dialogs** pull-down. The program processes the command value by calling WCLD (WinLoadDlg) to display the entry field dialog box. WCLD returns the dialog box handle hwndEntryField, which the program puts in hwndDlgBox. hwndDlgBox is then used by the routine CenterDlgBox, which positions the dialog box in the center of the screen.

Example Code: Entry Field Dialog Template

The entry field dialog box asks the user to enter two integer values in the range 1 through 100.

```
DLGTEMPLATE DLG_ENTRYFIELD LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Entry Field Example", DLG_ENTRYFIELD, 7, 40, 146, 112,
           WS_CLIPSIBLINGS | WS_SAVEBITS,
           FCF_DLGBORDER | FCF_NOBYTEALIGN | FCF_TITLEBAR
    BEGIN
        CONTROL "Enter data into the two entry ", ID_NULL, 6, 100, 124, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "fields below and press ENTER.", ID_NULL, 6, 92, 136, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "The program will then verify ", ID_NULL, 6, 78, 118, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "that fields contain integer", ID_NULL, 6, 70, 118, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "values in the range 1-100.", ID_NULL, 6, 62, 118, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "Entry Field 1 ...", ID_NULL, 20, 40, 70, 10, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "Entry Field 2 ...", ID_NULL, 20, 28, 70, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "", EF_1, 92, 42, 32, 8, WC_ENTRYFIELD, ES_LEFT | ES_AUTOSCROLL |
                ES_MARGIN | WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "", EF_2, 92, 28, 32, 8, WC_ENTRYFIELD, ES_LEFT | ES_AUTOSCROLL |
                ES_MARGIN | WS_TABSTOP | WS_VISIBLE
        CONTROL "Enter", 1, 6, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON | BS_DEFAULT |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", 2, 54, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_VISIBLE
        CONTROL "Help", HELP_ENTRYFIELD, 102, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
```

```
                    BS_HELP | WS_VISIBLE
    END
END
```

Input to the dialog box is processed by the dialog routine MyEntryFieldDlgRoutine, described in "MyEntryFieldDlgRoutine Walk Through" on page 12-28.

The MI-AUTORADIOBUTTON Command Value

The MI-AUTORADIOBUTTON command value occurs when the user selects the **Auto Radio button example...** choice from the **Dialogs** pull-down. ClientCommand processes the command value by calling WCLD (WinLoadDlg) to display the auto radio button dialog box. WCLD returns the dialog box handle hwndAutoRadioButton, which the program passes to the routine CenterDlgBox.

Example Code: Auto Radio Button Dialog Template

The dialog box displays a preview window and three radio buttons corresponding to the colors red, green, or blue.

```
DLGTEMPLATE DLG_AUTORADIOBUTTON LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Auto Radio Button Example", DLG_AUTORADIOBUTTON, 3, 45, 133, 122,
           WS_CLIPSIBLINGS | WS_SAVEBITS,
           FCF_DLGBORDER | FCF_NOBYTEALIGN | FCF_TITLEBAR
    BEGIN
        CONTROL "Select one of the radio", ID_NULL, 12, 108, 101, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "buttons, and the color", ID_NULL, 12, 100, 101, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "sample window will be", ID_NULL, 12, 92, 102, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "repainted in the color of", ID_NULL, 12, 84, 109, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "your choice.", ID_NULL, 12, 76, 54, 8, WC_STATIC, SS_TEXT |
                DT_CENTER | DT_TOP | WS_VISIBLE
        WINDOW  "", RECTL_AUTORADIOBUTTON, 73, 37, 32, 30, "Preview",
                WS_VISIBLE
        CONTROL "Color Preview", ID_NULL, 58, 25, 63, 8, WC_STATIC, SS_TEXT |
                DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "~Red", RB_RED, 6, 54, 38, 11, WC_BUTTON, BS_AUTORADIOBUTTON |
                WS_TABSTOP | WS_GROUP | WS_VISIBLE
        CONTROL "~Green", RB_GREEN, 6, 44, 41, 10, WC_BUTTON, BS_AUTORADIOBUTTON |
                WS_VISIBLE
        CONTROL "~Blue", RB_BLUE, 6, 34, 38, 10, WC_BUTTON, BS_AUTORADIOBUTTON |
                WS_VISIBLE
        CONTROL "Enter", 1, 6, 4, 36, 12, WC_BUTTON, BS_PUSHBUTTON | BS_DEFAULT |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", 2, 45, 4, 36, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_VISIBLE
        CONTROL "Help", HELP_AUTORADIOBUTTON, 86, 4, 35, 12, WC_BUTTON,
                BS_PUSHBUTTON | BS_HELP | WS_VISIBLE
```

```
    END
END
```

ClientCommand assigns the value of clrSample to clrPreview. This initializes the color of the preview window with the color of the sample window. It does this because the sample window color may have been changed by the user interacting with the check box example dialog box. The call WIQHID (WinWindowFromID) is used to find out the preview window handle from its symbolic identifier RECTL-AUTORADIOBUTTON. The call WINVRG (WinInvalidateRegion) sends a WM-PAINT message to the preview window. The message is posted on to the window as a WM-PPAINT message by the language-support window procedure. The message is passed to the preview window routine in the message-processing loop, and the routine processes the message by repainting the window.

The program does not explicitly have to create the preview window in either the auto radio button or check box dialog boxes. They are drawn by the system, because they are specified as child windows of their respective dialog boxes in the .DLG file.

Compare this with the color sample window, which was explicitly created by a call to WICRT (WinCreateWindow).

Both color preview windows were added to their respective dialog boxes using a text editor.

Input to the dialog box is processed by the dialog routine MyAutoRadioButtonDlgRoutine, described in "MyAutoRadioButtonDlgRoutine Walk Through" on page 12-18.

The MI-CHECKBOX Command Value

The MI-CHECKBOX command value occurs when the user selects the **Check box example...** choice from the **Dialogs** pull-down. ClientCommand processes the command value by calling WCLD (WinLoadDlg) to display the check box dialog box. WCLD returns the dialog box handle hwndCheckBox, which the program passes to the routine CenterDlgBox.

Example Code: Check Box Dialog Template

The check box dialog box displays a preview window and three check boxes, corresponding to the colors red, green, and blue.

```
DLGTEMPLATE DLG_CHECKBOX LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Check Box Example", DLG_CHECKBOX, 3, 38, 138, 115,
           WS_CLIPSIBLINGS | WS_SAVEBITS,
           FCF_DLGBORDER | FCF_NOBYTEALIGN | FCF_TITLEBAR
    BEGIN
        CONTROL "Select one or more check", ID_NULL, 6, 101, 122, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "boxes, and the color sample", ID_NULL, 6, 93, 123, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "window will be painted in the", ID_NULL, 6, 85, 128, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "color(s) of your choice.", ID_NULL, 6, 77, 103, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        WINDOW  "", RECTL_CHECKBOX, 70, 36, 38, 28, "Preview", WS_VISIBLE
        CONTROL "Color Preview", ID_NULL, 58, 24, 63, 8, WC_STATIC, SS_TEXT |
                DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "~Red", CB_RED, 6, 55, 42, 10, WC_BUTTON, BS_CHECKBOX |
                WS_TABSTOP | WS_GROUP | WS_VISIBLE
        CONTROL "~Green", CB_GREEN, 6, 45, 42, 10, WC_BUTTON, BS_CHECKBOX |
                WS_VISIBLE
```

```
        CONTROL "~Blue", CB_BLUE, 6, 35, 42, 10, WC_BUTTON, BS_CHECKBOX |
                WS_VISIBLE
        CONTROL "Enter", 1, 6, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON | BS_DEFAULT |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", 2, 50, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_VISIBLE
        CONTROL "Help", HELP_CHECKBOX, 94, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                BS_HELP | WS_VISIBLE
    END
END
```

ClientCommand moves the color of the sample window from clrSample to clrPreview, the color of the preview window.

The call WUSVAL (WinSetValue) is used to set mp1Value to TRUE-VALUE.

ClientCommand then determines the constituent colors of the current color of the preview window. Three separate occurrences of the call WUQBM (WinQueryBitsUnderMask) are used to extract the bits that respectively correspond to CLR-RED, CLR-GREEN, and CLR-BLUE from the preview window color. The call uses the CLR- constant itself as the bit mask. Three separate occurrences of the call WIQHID (WinQueryWindowFromID) are used to obtain the handle of the appropriate check box from its identifier, and three separate occurrences of the call WMPOST (WinPostMsg) post a BM-SETCHECK message to the red, blue, or green check boxes to tell them to set themselves to the checked status that match the constituent colors. The message uses the mp1Value set by the earlier WUSVAL (WinSetValue).

Finally, a further call to WIQHID (WinWindowFromID) is used to find out the preview window handle from its symbolic identifier RECTL-AUTORADIOBUTTON. The call WINVRG (WinInvalidateRegion) sends a WM-PAINT message to the preview window. The message is posted on to the window as a WM-PPAINT by the language-support window procedure. The message is passed to the preview window routine in the message-processing loop, and the routine processes the message by repainting the window.

Input to the dialog box is processed by the dialog routine MyCheckBoxDlgRoutine, described in "MyCheckBoxDlgRoutine Walk Through" on page 12-22.

The MI-LISTBOX Command Value

The MI-LISTBOX command value occurs when the user selects the **List box example...** choice from the **Dialogs** pull-down. ClientCommand processes the command value by calling WCLD (WinLoadDlg) to display the list box dialog box. WCLD returns the dialog box handle hwndListBox, which the program passes to the routine CenterDlgBox.

Example Text: List Box Dialog Template

The list box dialog box displays a list of text strings, and allows a single selection from the list.

```
DLGTEMPLATE DLG_LISTBOX LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "List Box Example", DLG_LISTBOX, 5, 36, 146, 105,
           WS_CLIPSIBLINGS | WS_SAVEBITS,
           FCF_DLGBORDER | FCF_NOBYTEALIGN | FCF_TITLEBAR
    BEGIN
        CONTROL "Select an item and press Enter", ID_NULL, 6, 93, 139, 8,
                WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_VISIBLE
        CONTROL "", LISTBOX, 25, 26, 98, 58, WC_LISTBOX, WS_TABSTOP | WS_GROUP
```

```
                | WS_VISIBLE
        CONTROL "Enter", 1, 6, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON | BS_DEFAULT |
                WS_GROUP | WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", 2, 54, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                WS_VISIBLE
        CONTROL "Help", HELP_LISTBOX, 102, 4, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                BS_HELP | WS_VISIBLE
    END
END
```

The call WUSVAL (WinSetValue) is used to set mp1Value to LIT-END, which is the index value of the end of the list of items in the list box. ClientCommand calls the Initialize-Listbox routine to initialize the list box with the strings LB_1 through LB_24 from the string table in the program's resource file.

Example Code: Initialize Listbox Routine

```
*****************************************************************************
**                                                                         *
** PRIVATE FUNCTION : Initialize-ListBox                                   *
**                                                                         *
** Puts a string loaded from a resource file into a listbox.               *
**                                                                         *
*****************************************************************************

 Initialize-Listbox.

* Load from stringtable into intermediate variable Buffer
     compute idString = LBI-1 + i
     move LEN-LISTBOXENTRY to MaxStringLength
     call "WMLSTR" using  hab,
                          NULL,
                          idString,
                          MaxStringLength,
                          Buffer,
                          BufferSize

* Create string for mp2 (contents are different each time through loop)
         call "WUCDS" using  hab,
                             BufferCount,
                             BufferStringTypes,
                             BufferSize,
                             Buffer,
                             BufferMaxSize,
                             BufferHandle,
                             dummy

* Add item to the listbox
         call "WCSDIM" using  hwndListBox,
                              LISTBOX,
                              LM-INSERTITEM,
                              mp1Value,
                              BufferHandle,
                              dummy
```

```
* Destroy the data structure
        call "WUDDS" using  hab,
                            BufferHandle,
                            dummy.

 EndInitialize-Listbox.
```

Initialize-Listbox calls WMLSTR (WinLoadString) to load a string, and calls WCSDIM (WinSendDlgItemMsg) to send an LM-INSERTITEM message to the list box to insert the item. The message uses the LIT-END index value to insert each loaded message at the end of the list. The second parameter of the LM-INSERTITEM message is a handle to the text that you want to insert. In COBOL and FORTRAN, when a PM call has a handle of data type HSTRUCT an input or output parameter, you must use the call WUCDS (WinCreateDataStructure) to create a data structure in the form you require. In the third parameter of the call, you specify the structure by using standard data types. They can be simple data types, structure data types, pointer data types, control data types, or any valid combination. The sample program is only specifiying a string, so uses data type DTYP-STRL in the parameter BufferTypes.

The call returns the handle of the structure to the program for subsequent use. In the sample, WUCDS creates a data structure and places the contents of Buffer in the PM structure identified by the handle returned in BufferHandle. The handle is used in the call WCSDIM (WinSendDlgItemMsg) that sends the LM-INSERTITEM message to the list box. As soon as your program has finished using a structure created by WUCDS, it must call WUDDS (WinDestroyDataStructure) to destroy it. See "Using Data Structures" on page 12-35 for more details about structures.

When all the strings have been loaded into the list box, ClientCommand calls WUSVAL (WinSetValue) twice to set the message parameters used by the subsequent WCSDIM (WinSendDlgItemMsg). The latter sends an LM-SELECTITEM message to the list box to select the first item.

Input to the dialog box is processed by the dialog routine MyListBoxDlgRoutine, described in "MyListBoxDlgRoutine Walk Through" on page 12-34.

Dialog Routine MyAutoRadioButtonDlgRoutine

```
*************************************************************************
*                    Start of First Dialog Routine                      *
*************************************************************************
*************************************************************************
**                                                                      *
** Dialog Routine : MyAutoRadioButtonDlgRoutine                         *
**                                                                      *
** A dialog routine that uses auto radio buttons to change the         *
** color of the Preview window and the Color Sample window.            *
** One color only can therefore be selected.                           *
**                                                                      *
*************************************************************************

 MyAutoRadioButtonDlgRoutine.

* Switch on message type

* If Help message
     if (QMSG-MSGID = WM-HELP)
```

```
* Call private help function

       perform HelpProc through EndHelpProc

* Else If Control message
     else if (QMSG-MSGID = WM-PCONTROL)

*************************************************************************
* Call Query Value to extract control and new color
* values from QMSG-PARAM1.
* The control value is contained in the high word of QMSG-PARAM1.
* WUQVAL extracts elements in high order sequence, so
* the first returned value contains the control, and the
* second contains the new color.
*************************************************************************

       move 2 to ValuesCount
       call "WUQVAL" using  hab,
                            QMSG-PARAM1,
                            ValuesCount,
                            mp1ValueTypes,
                            mp1ValueData,
                            dummy

       move mp1Values(1) to idControl
       move mp1Values(2) to NewColor

       if (idControl = BN-CLICKED)

* Determine Preview window new color from the selected auto radio button

         if (NewColor = RB-RED)
           move CLR-RED to clrPreview
         else
           if (NewColor = RB-GREEN)
             move CLR-GREEN to clrPreview
           else
             if (NewColor = RB-BLUE)
               move CLR-BLUE to clrPreview
             else
* Else Return from dialog routine
               go to EndAutoRadioButtonDlg

* End of Switch on New Color
             end-if
           end-if
        end-if
* End of Switch on Control type
       end-if

* Force repaint of the Preview window in the new color
       call "WINVRG" using  hwndPreview,
                            NULL,
                            FALSE-VALUE,
                            dummy

* Else If Command message
    else if (QMSG-MSGID = WM-COMMAND)
```

```
*************************************************************************
* Call Query Value to extract command value from QMSG-PARAM1.
* The required value is contained in the low word of QMSG-PARAM1.
* WUQVAL extracts elements in high order sequence, so
* it is the second returned value that is required.
*************************************************************************

      move 2 to ValuesCount
      call "WUQVAL" using  hab,
                           QMSG-PARAM1,
                           ValuesCount,
                           mp1ValueTypes,
                           mp1ValueData,
                           dummy

      move mp1Values(2) to idCommand

* If enter key or pushbutton pressed or selected
      if (idCommand = DID-OK)

* Set color to Preview color
        move clrPreview to clrSample

* Force repaint of Color Sample window
        call "WINVRG" using  hwndColorSample,
                             NULL,
                             FALSE-VALUE,
                             dummy

* Else if Escape key or CANCEL pushbutton pressed or selected
      else
        if (idCommand = DID-CANCEL)

* Do Nothing
        continue

* Else message is passed to default dialog procedure

        else
          call "WCDDP" using  QMSG-HWND,
                              QMSG-MSGID,
                              QMSG-PARAM1,
                              QMSG-PARAM2,
                              dummy

* go to end of dialog routine
          go to EndAutoRadioButtonDlg

* End of Switch on Command type
        end-if
      end-if

* Finished with dialog box
      call "WIDEL" using  hwndAutoRadioButton,
                          dummy

* Repaint Area
      call "WINVRG" using  hwndClient,
                           NULL,
                           FALSE-VALUE,
```

```
                                dummy

* Set dialog box handle to null
        move 0 to hwndAutoRadioButton

* Else message is passed to default dialog procedure
    else
      call "WCDDP" using  QMSG-HWND,
                          QMSG-MSGID,
                          QMSG-PARAM1,
                          QMSG-PARAM2,
                          dummy

* go to end of dialog routine
       go to EndAutoRadioButtonDlg

* End of Switch on message type
    end-if.

 EndAutoRadioButtonDlg.

*****************************************************************************
*                    End of First Dialog Routine                            *
*****************************************************************************
```

MyAutoRadioButtonDlgRoutine Walk Through

When the message-processing loop in the main procedure encounters a message destination of hwndAutoRadioButton, it calls the dialog routine MyAutoRadioButtonDlgRoutine. The routine processes the WM-HELP, WM-PCONTROL, and WM-COMMAND messages.

The WM-HELP Message

The WM-HELP message is processed by calling the function HelpProc.

The WM-PCONTROL Message

The dialog routine monitors user interaction with the radio buttons *before* the **Enter** pushbutton is selected, and changes the color of the preview window to match the button selected, while the dialog box is still displayed. (Compare this with the entry field example dialog procedure, which acts on the contents of the dialog box only when the user selects **Enter**, as indicated by the WM-COMMAND message.)

Because the radio buttons are defined in the dialog template as BS_AUTORADIOBUTTON, the program does not have to check or uncheck the radio buttons. The system checks and unchecks them. When the user clicks on an auto radio button, its system-provided window procedure paints the button black to show it is selected, deselects the other radio buttons, and sends a WM-CONTROL message to its owner, the dialog box. The message is intercepted by the language-support dialog procedure, and posted on to the program.

The first parameter of the message contains the identifier of the radio button (RB-RED, RB-GREEN, or RB-BLUE) and a notification code. The routine calls WUQVAL (WinQueryValue) to extract the identifier and the notification code from the message and place them in the mp1ValueData array. The identifier of the button is placed in NewColor. The notification code is placed in idControl.

If the notification code is BN-CLICKED, the user has clicked on a button. The program tests the identifier of the button to discover which button was clicked, and sets the color clrPreview to CLR-RED, CLR-GREEN, or CLR-BLUE accordingly.

The call WINVRG (WinInvalidateRegion) sends a WM-PAINT message to the preview window so it

can repaint itself in the new color. Because most of the rest of the dialog routine consists of the WM-COMMAND processing, the dialog box remains on the screen after the dialog routine has processed the WM-PCONTROL message. This enables the user to select a different radio button and see the result in the preview window any number of times. The dialog box remains on the screen until the user clicks on the **Enter** or **Cancel** pushbuttons.

The WM-COMMAND Message

When the user selects a pushbutton, its system-provided window procedure posts a WM-COMMAND message to its owner, the dialog box. The first parameter of the message contains the command value of the pushbutton, DID-OK for **Enter**, and DID-CANCEL for **Cancel**. The dialog routine calls WUQVAL (WinQueryValue) to extract the command value, and places it in idCommand.

DID-OK Command Value

If the command value is DID-OK, the program sets the color of the sample window to the color of the preview window, and calls WINVRG (WinInvalidateRegion) to send a WM-PAINT message to the sample window so that it can repaint itself.

DID-CANCEL Command Value

If the command value is DID-CANCEL, the routine does nothing, and processing drops through to the WIDEL (WinDestroyWindow) call. The program follows the practice of passing messages containing any other command values to the default window procedure. It then goes to the end of the routine.

Destroying the Dialog Box and Repainting the Client

Whether the WM-COMMAND was for **Enter** or **Cancel**, the program has to remove the dialog box from the screen. To do this, it calls WIDEL (WinDestroyWindow). A call to WINVRG (WinInvalidateRegion) sends a WM-PAINT to the client window to tell it to repaint itself. The program sets the dialog box handle hwndAutoRadioButton to zero, as the system gives the dialog box a unique handle each time it is created.

Default Dialog Processing

The program passes unprocessed messages to WCDDP (WinDefDlgProc), the default dialog procedure.

Dialog Routine MyCheckBoxDlgRoutine

```
*************************************************************************
*                  Start of Second Dialog Routine                       *
*************************************************************************
*************************************************************************
**                                                                      *
** Dialog Routine : MyCheckBoxDlgRoutine                                *
**                                                                      *
** A dialog routine which uses checkboxes to change the color          *
** of the Preview window and the Color Sample window.                   *
** One or more color(s) can therefore be selected.                      *
**                                                                      *
*************************************************************************

 MyCheckBoxDlgRoutine.

* Switch on message type

* If Help message
```

```
    if (QMSG-MSGID = WM-HELP)

* Invoke private help function

      perform HelpProc through EndHelpProc

* Else If Control message
    else if (QMSG-MSGID = WM-PCONTROL)

*************************************************************************
* Call Query Value to extract control and color
* values from QMSG-PARAM1.
* The control value is contained in the high word of QMSG-PARAM1.
* WUQVAL extracts elements in high order sequence, so
* the first returned value contains the control, and the second
* returns the color.
*************************************************************************

      move 2 to ValuesCount
      call "WUQVAL" using  hab,
                           QMSG-PARAM1,
                           ValuesCount,
                           mp1ValueTypes,
                           mp1ValueData,
                           dummy

      move mp1Values(1) to idControl
      move mp1Values(2) to idDlgItem

* If user has clicked on a checkbox
      if (idControl = BN-CLICKED)

* This may be a checkbox selection or deselection - add or subtract
* the latest color change

        perform CheckColor through End-CheckColor

* Obtain handle of Preview window

        call "WIQHID" using
                     hwndCheckBox,
                     RECTL-CHECKBOX,
                     hwndPreview

* Force repaint of preview window in the new color(s)

        call "WINVRG" using  hwndPreview,
                             NULL,
                             FALSE-VALUE,
                             dummy

* End if user has clicked on a checkbox
      end-if

* Else If Command message

    else if (QMSG-MSGID = WM-COMMAND)

*************************************************************************
* Call Query Value to extract command value from QMSG-PARAM1.
```

```
* The required value is contained in the low word of QMSG-PARAM1.
* WUQVAL extracts elements in high order sequence, so
* it is the second returned value that is required.
*************************************************************************

       move 2 to ValuesCount
       call "WUQVAL" using  hab,
                            QMSG-PARAM1,
                            ValuesCount,
                            mp1ValueTypes,
                            mp1ValueData,
                            dummy

       move mp1Values(2) to idCommand

* If enter key or pushbutton pressed or selected
         if (idCommand = DID-OK)

* Force repaint of color sample window in new color(s)
           move clrPreview to clrSample

           call "WINVRG" using  hwndColorSample,
                                NULL,
                                FALSE-VALUE,
                                dummy

* Else if Escape key or CANCEL pushbutton pressed or selected
         else
           if (idCommand = DID-CANCEL)

* Do Nothing
           continue

* Else message is passed to default dialog procedure
           else
             call "WCDDP" using  QMSG-HWND,
                                 QMSG-MSGID,
                                 QMSG-PARAM1,
                                 QMSG-PARAM2,
                                 dummy

* go to end of dialog routine
             go to EndCheckBoxDlg

* End of Switch on Command type
           end-if
         end-if

* Finished with dialog box
         call "WIDEL" using  hwndCheckBox,
                             dummy

* Repaint Area
         call "WINVRG" using  hwndClient,
                              NULL,
                              FALSE-VALUE,
                              dummy

* Set dialog box handle to null
         move 0 to hwndCheckbox
```

```
* go to end of dialog routine
        go to EndCheckBoxDlg


* Else message is passed to default dialog procedure
    else
      call "WCDDP" using  QMSG-HWND,
                          QMSG-MSGID,
                          QMSG-PARAM1,
                          QMSG-PARAM2,
                          dummy

* End of Switch on message type
    end-if.

 EndCheckBoxDlg.

*****************************************************************************
*                    End of Second Dialog Routine                           *
*****************************************************************************
```

MyCheckBoxDlgRoutine Walk Through

When the message-processing loop in the main procedure encounters a message destination of hwndCheckBox, it calls the dialog routine MyCheckBoxDlgRoutine. The routine processes the WM-HELP, WM-PCONTROL, and WM-COMMAND messages.

The WM-HELP Message

The WM-HELP message is processed by calling the function HelpProc.

The WM-PCONTROL Message

Similarly to the radio button dialog, the check box dialog routine monitors user interaction with the check boxes *before* the **Enter** pushbutton is selected, and changes the color of the preview window to match the check boxes selected, while the dialog box is still displayed.

However, because the check boxes are defined as BS_CHECKBOX in the dialog template (see "Example Code: Check Box Dialog Template" on page 12-12), rather than BS_AUTOCHECKBOX, the program has to check or uncheck a check box when the user clicks it. Another difference is that the radio button dialog allows the user to select a single color only, and displays the chosen single color in its preview window. The check box dialog allows the user to select any combination of the colors red, green, and blue and displays the chosen color mix in its preview window.

When the user clicks on a check box, its system-provided window procedure sends a WM-CONTROL message to its owner, the dialog box. The message is intercepted by the language-support dialog procedure, and posted on to the program.

The first parameter of the message contains the identifier of the check box (CB-RED, CB-GREEN, or CB-BLUE) and a notification code. The routine calls WUQVAL (WinQueryValue) to extract the identifier and the notification code from the message and place them in the mp1ValueData array. The identifier of the button is placed in idDlgItem. The notification code is placed in idControl.

If the notification code is BN-CLICKED, the user has clicked on a check box. The user may be selecting or deselecting the check box. The dialog routine calls the CheckColor routine to add the selected color to, or remove the selected color from, the color that is to be displayed in the preview window.

CheckColor Routine

```
*****************************************************************************
**                                                                         *
** PRIVATE FUNCTION : CheckColor                                           *
**                                                                         *
** Toggle the Checked/UnChecked state of a checkbox and add/remove         *
** the corresponding CLR color component of the Preview window color.      *
**                                                                         *
*****************************************************************************

 CheckColor.

* Switch on idDlgItem
* Set color according to checkbox ID value
     if (idDlgItem = CB-RED)
       move CLR-RED to clr

     else
       if (idDlgItem = CB-GREEN)
         move CLR-GREEN to clr

       else
         if (idDlgItem = CB-BLUE)
         move CLR-BLUE to clr

* Else go to end of procedure

       else
         go to End-CheckColor

         end-if
       end-if
     end-if

* Obtain handle of checkbox window
     call "WIQHID" using  hwndCheckBox,
                          idDlgItem,
                          hwndCheck

* Query the checked or unchecked state of the checkbox
     call "WMSEND" using  hwndCheck,
                          BM-QUERYCHECK,
                          NULL,
                          NULL,
                          fChecked

* Set or reset the checkbox state, to the reverse of fChecked
     if (fChecked not = FALSE-VALUE)
       move FALSE-VALUE to NewState
     else
       move TRUE-VALUE  to NewState
     end-if

* Set Value for mp1
     move 0        to mp1Values(1)
     move NewState to mp1Values(2)
     move 2 to ValuesCount
```

```
        call "WUSVAL" using  hab,
                             ValuesCount,
                             mp1ValueTypes,
                             mp1ValueData,
                             mp1Value,
                             dummy

    * Set Value for mp2
        move 0 to mp2Values(1)
        move 0 to mp2Values(2)

        call "WUSVAL" using  hab,
                             ValuesCount,
                             mp2ValueTypes,
                             mp2ValueData,
                             mp2Value,
                             dummy

    * Obtain handle of checkbox window
        call "WIQHID" using  hwndCheckBox,
                             idDlgItem,
                             hwndCheck

    * Post message to Checkbox window
        call "WMPOST" using  hwndCheck,
                             BM-SETCHECK,
                             mp1Value,
                             mp2Value,
                             dummy

    * Add or subtract the checked or unchecked color choice
        if (fChecked not = FALSE-VALUE)
          compute clrPreview = (clrPreview - clr)
        else
          compute clrPreview = (clrPreview + clr)
        end-if.

      End-CheckColor.
```

CheckColor Routine Walk Through

CheckColor tests idDlgItem to find out which button was clicked, and puts the corresponding color in the variable clr. The call WIQHID (WinWindowFromID) obtains the handle of the clicked check box, given its identifier. The call WMSEND (WinSendMsg) sends a BM-QUERYCHECK message to the check box identified by the window handle, to discover if it is checked or unchecked. The two calls WIQHID (WinWindowFromID) and WMSEND (WinSendMsg) could be replaced by a single call to WCSDIM (WinSendDlgItemMsg) which is a functional equivalent of the WIQHID and WMSEND calls.

The routine sets NewState to the reverse of the current setting of the check box. NewState is used in the BM-SETCHECK that is posted to the check box. Two calls to WUSVAL (WinSetValue) compose the message parameters used in the subsequent WMPOST (WinPostMsg). Finally, CheckColor adds the color to, or subtracts it from, the color of the preview window.

Back in the check box dialog routine, a call to WIQHID (WinWindowFromID) obtains the handle of the preview window, and a call to WINVRG (WinInvalidateRegion) sends a WM-PAINT message to the preview window to tell it to repaint itself.

The WM-COMMAND Message

In the check box dialog routine, when the user selects a pushbutton, its system-provided window procedure posts a WM-COMMAND message to its owner, the dialog box. The routine processes the message in exactly the same way as the radio button dialog described in "The WM-COMMAND Message" on page 12-19.

Default Dialog Processing

The program passes unprocessed messages to WCDDP (WinDefDlgProc), the default dialog procedure.

Dialog Routine MyEntryFieldDlgRoutine

```
*************************************************************************
*                    Start of Third Dialog Routine                      *
*************************************************************************
*************************************************************************
**                                                                      *
** Dialog Routine : MyEntryFieldDlgRoutine                              *
**                                                                      *
** A dialog routine which captures and validates the contents of        *
** two entry fields.                                                    *
**                                                                      *
*************************************************************************

 MyEntryFieldDlgRoutine.

* Switch on message type

* If Help message
     if (QMSG-MSGID = WM-HELP)

* Call private help function

       perform HelpProc through EndHelpProc

* Else if Command message
     else if (QMSG-MSGID = WM-COMMAND)

*************************************************************************
* Call Query Value to extract command value from QMSG-PARAM1.
* The required value is contained in the low word of QMSG-PARAM1.
* WUQVAL extracts elements in high order sequence, so
* it is the second returned value that is required.
*************************************************************************

       move 2 to ValuesCount
       call "WUQVAL" using  hab,
                            QMSG-PARAM1,
```

```
                              ValuesCount,
                              mp1ValueTypes,
                              mp1ValueData,
                              dummy

       move mp1Values(2) to idCommand

* If Enter key or pushbutton pressed or selected

       if (idCommand = DID-OK)

*************************************************************************
*                                                                       *
*  Validate the contents of the two entry fields, and                   *
*  if either entry field is not valid, return leaving                   *
*  dialog box visible (by not dismissing it).                           *
*                                                                       *
*************************************************************************

        move EF-1 to idEntryField
        perform IsIntInRange through EndIsIntInRange
        if (ValueInRange not = TRUE-VALUE)
          go to EndEntryFieldDlg
        end-if

        move EF-2 to idEntryField
        perform IsIntInRange through EndIsIntInRange
        if (ValueInRange not = TRUE-VALUE)
          go to EndEntryFieldDlg
        end-if

*************************************************************************
*                                                                       *
* Both entry fields are valid - save their contents.                    *
* (A modal dialog is destroyed before control is returned               *
* to the invoking window routine, so it must pass the contents          *
* of its Entry Fields etc back to the invoking window                   *
* before it returns.)                                                   *
*                                                                       *
*************************************************************************

* Obtain handles of EntryFields 1 and 2

        call "WIQHID" using  hwndEntryField,
                             EF-1,
                             hwndEF1

        call "WIQHID" using  hwndEntryField,
                             EF-2,
                             hwndEF2

* Query window text for each entryfield
        move 9 to MaxStringLength
        call "WIQTXT" using  hwndEF1,
                             MaxStringLength,
                             EntryField1,
                             EntryField1Size

        call "WIQTXT" using  hwndEF2,
                             MaxStringLength,
```

```
                                EntryField2,
                                EntryField2Size

           call "WIDEL" using  hwndEntryField,
                               dummy

           call "WINVRG" using  hwndClient,
                                NULL,
                                FALSE-VALUE,
                                dummy

* Set dialog box handle to null
           move 0 to hwndEntryField


* Else if escape key or CANCEL pushbutton pressed or selected
         else
           if (idCommand = DID-CANCEL)

* Finished with dialog box so destroy window and force a repaint
           call "WIDEL" using  hwndEntryField,
                               dummy

           call "WINVRG" using  hwndClient,
                                NULL,
                                FALSE-VALUE,
                                dummy

* Set dialog box handle to null
           move 0 to hwndEntryField

* End of Switch on Command type
           end-if
         end-if

* Else message is passed to default dialog procedure
       else

         call "WCDDP" using  QMSG-HWND,
                             QMSG-MSGID,
                             QMSG-PARAM1,
                             QMSG-PARAM2,
                             dummy

* End of Switch on message type
       end-if.

 EndEntryFieldDlg.

*****************************************************************************
*                    End of Third Dialog Routine                            *
*****************************************************************************
```

MyEntryFieldDlgRoutine Walk Through

The entry field dialog box asks the user to enter two integer values in the range 1 through 100. When the message-processing loop in the main procedure encounters a message destination of hwndEntryField, it calls the dialog routine MyEntryFieldDlgRoutine. The routine processes the WM-HELP, and WM-COMMAND messages.

The WM-HELP Message

The WM-HELP message is processed by calling the function HelpProc.

The WM-COMMAND Message

When the user selects a pushbutton, its system-provided window procedure posts a WM-COMMAND message to its owner, the dialog box. The first parameter of the message contains the command value of the pushbutton, DID-OK for **Enter**, and DID-CANCEL for **Cancel**. The dialog routine calls WUQVAL (WinQueryValue) to extract the command value, and places it in idCommand.

DID-OK Command Value

If the command value is DID-OK, for each entry field that routine puts its identifier in idEntryField and calls the routine IsIntInRange.

IsIntInRange Routine

```
 IsIntInRange.


*************************************************************************
*                                                                       *
*   Validate an entry field.                                            *
*                                                                       *
* If validation fails leave the dialog visible, issue an error         *
*  message using a messagebox, and when the user dismisses the         *
*  messagebox, set the input focus to the entry field containing       *
*  the error. Leave the contents of the entry field unchanged,         *
*  and set ValueInRange to FALSE.                                      *
*                                                                       *
* If validation is successful return the value in Value and            *
*  set ValueInRange to TRUE.                                           *
*                                                                       *
*                                                                       *
* Indicate an error if either:                                         *
*      1. Conversion of input text to integer is unsuccessful          *
*  or  2. Value is less than LoRange                                   *
*  or  3. Value is greater than HiRange                                *
*************************************************************************

* Initialize data
     move 1   to LoRange
     move 100 to HiRange

     move ERR-EFINVALID to idErrMsg
     move ID-MB         to idMessageBox

     call "WCQDIS" using  hwndEntryField,
                          idEntryField,
                          Value-Entered,
                          TRUE-VALUE,
                          Success
```

```
    if ( (Value-Entered = FALSE-VALUE) OR
         (Value-Entered < LoRange) OR
         (Value-Entered > HiRange) )

* Load error message
      move 80 to MaxStringLength
      call "WMLSTR" using  hab,
                           NULL,
                           idErrMsg,
                           MaxStringLength,
                           ErrMsg,
                           ErrMsgSize

* Display a message box with MB-ENTER and CUA warning style
      compute MessageBoxStyle = MB-ENTER + MB-CUAWARNING.

      call "WCMBOX" using  HWND-DESKTOP,
                           hwndEntryField,
                           ErrMsgSize,
                           ErrMsg,
                           NULL-PTR,
                           EMPTY-STR,
                           idMessageBox,
                           MessageBoxStyle,
                           dummy

* Obtain handle of entryfield in error
      call "WIQHID" using  hwndEntryField,
                           idEntryField,
                           EntryFieldHandle

* Set the input focus to the entry field in error
      call "WMSFOC" using  HWND-DESKTOP,
                           EntryFieldHandle,
                           dummy

* Set ValueInRange to FALSE
      move FALSE-VALUE to ValueInRange

* Else Set ValueInRange to TRUE
    else
      move TRUE-VALUE to ValueInRange

    end-if.

 EndIsIntInRange.
```

IsIntInRange Walk Through

Because the contents of each entry field are in ASCII text, IsIntInRange calls WCQDIS (WinQueryDlgItemShort) to convert the text to binary, and tests that the result is in the range 1 through 100.

If the number is not in range, the routine initializes MessageBoxStyle with the MB-ENTER style, and calls WUSBM (WinSetBitsUnderMask) to modify MessageBoxStyle with the MB-CUAWARNING style. The call WCMBOX (WinMessageBox) displays the message box. The call WIQHID (WinQueryWindowID) obtains the handle of the entry field in error, calls WMSFOC (WinSetFocus) to set the input focus to the field so that the user

can enter a correct number, and sets the variable ValueInRange to FALSE-VALUE.

If the number is in range, it sets ValueInRange to TRUE-VALUE.

Back in the dialog routine, the routine goes to the end of the routine if a false value was found by IsIntInRange. If the values were okay, two calls to WIQHID (WInWindowFromID) obtain the handles of the two entry fields, and two calls to WIQTXT (WinQueryWindowText) copy the integers from each entry field to the buffers EntryField1 and EntryField2. The routine calls WIDEL (WinDestroyWindow) to remove the dialog box, calls WINVRG (WinInvalidateRegion) to send a WM-PAINT message to the client window, and sets the dialog box handle to zero.

DID-CANCEL Command Value

If the command value is DID-CANCEL, the routine calls WIDEL (WinDestroyWindow) for the dialog box, and WINVRG (WinInvalidateRegion) for the client window, and sets the dialog box handle to zero.

Destroying the Dialog Box and Repainting the Client

Whether the WM-COMMAND was for **Enter** or **Cancel**, the program has to remove the dialog box from the screen. To do this, it calls WIDEL (WinDestroyWindow). A call to WINVRG (WinInvalidateRegion) sends a WM-PAINT to the client window to tell it to repaint itself.

Default Dialog Processing

The program passes unprocessed messages to WCDDP (WinDefDlgProc), the default dialog procedure.

Dialog Routine MyListBoxDlgRoutine

```
*****************************************************************************
*                    Start of Fourth Dialog Routine                         *
*****************************************************************************
*****************************************************************************
**                                                                          *
** Dialog Routine : MyListBoxDlg                                            *
**                                                                          *
** A dialog routine that displays a list of text strings and               *
** permits a single selection from the list                                 *
**                                                                          *
*****************************************************************************

 MyListBoxDlgRoutine.

* Switch on message type

* If Help message
     if (QMSG-MSGID = WM-HELP)

* Call private help function

       perform HelpProc through EndHelpProc

* Else if Command message
     else if (QMSG-MSGID = WM-COMMAND)

*****************************************************************************
* Call Query Value to extract command value from QMSG-PARAM1.
```

```
* The required value is contained in the low word of QMSG-PARAM1.
* WUQVAL extracts elements in high order sequence, so
* it is the second returned value that is required.
*************************************************************************

       move 2 to ValuesCount
       call "WUQVAL" using  hab,
                            QMSG-PARAM1,
                            ValuesCount,
                            mp1ValueTypes,
                            mp1ValueData,
                            dummy

       move mp1Values(2) to idCommand

* If Enter key or pushbutton pressed or selected

       if (idCommand = DID-OK)

*************************************************************************
* Find out which item, if any, was selected and return the             *
*  selected item text.                                                 *
* (By default, only one item can be selected)                          *
*************************************************************************

         call "WCSDIM" using  hwndListBox,
                              LISTBOX,
                              LM-QUERYSELECTION,
                              NULL,
                              NULL,
                              Reply

*************************************************************************
* Call Query Value to extract ID from Reply.
* The required value is contained in the low word of Reply.
* WUQVAL extracts elements in high order sequence, so
* it is the second returned value that is required.
*************************************************************************

         call "WUQVAL" using  hab,
                              Reply,
                              ValuesCount,
                              ReplyValueTypes,
                              ReplyValueData,
                              dummy

         move ReplyValues(2) to idItem

* If id is None
         if (idItem = LIT-NONE)

* Nothing has been selected so set Selection to blank
           move 0 to SelectionSize

         else
* Set new value for mp1
           move LEN-LISTBOXENTRY to mp1Values(1)
           move idItem to mp1Values(2)

           call "WUSVAL" using  hab,
```

```
                                    ValuesCount,
                                    mp1ValueTypes,
                                    mp1ValueData,
                                    mp1Value,
                                    dummy

* Create data structure for string
          move 0 to SelectionSize
          call "WUCDS" using  hab,
                              SelectionCount,
                              SelectionStringTypes,
                              SelectionSize,
                              Selection,
                              SelectionMaxSize,
                              SelectionHandle,
                              dummy

          call "WCSDIM" using  hwndListBox,
                               LISTBOX,
                               LM-QUERYITEMTEXT,
                               mp1Value,
                               SelectionHandle,
                               dummy

* Query and destroy the data structure
          move 15 to SelectionSize
          call "WUQDS" using  hab,
                              SelectionHandle,
                              SelectionCount,
                              SelectionStringTypes,
                              SelectionSize,
                              Selection,
                              SelectionSize,
                              dummy

          call "WUDDS" using  hab,
                              SelectionHandle,
                              dummy

* End If id is none
        end-if

* Finished with dialog box so destroy window and force a repaint
        call "WIDEL" using  hwndListBox,
                            dummy

        call "WINVRG" using  hwndClient,
                             NULL,
                             FALSE-VALUE,
                             dummy

* Set dialog box handle to null
        move 0 to hwndListBox

* Else if escape key or CANCEL pushbutton pressed or selected
      else
        if (idCommand = DID-CANCEL)

* Finished with dialog box so destroy window and force a repaint
          call "WIDEL" using  hwndListBox,
```

```
                                dummy

            call "WINVRG" using  hwndClient,
                                 NULL,
                                 FALSE-VALUE,
                                 dummy

* Set dialog box handle to null
            move 0 to hwndListBox

* go to end of dialog routine
            go to EndListBoxDlg

* End of Switch on Command type
          end-if
        end-if

* Else If Control message
      else if (QMSG-MSGID = WM-PCONTROL)

*************************************************************************
* Call Query Value to extract control value from QMSG-PARAM1.
* The control value is contained in the high word of QMSG-PARAM1.
* WUQVAL extracts elements in high order sequence, so
* the first returned value contains the control.
*************************************************************************

        move 2 to ValuesCount
        call "WUQVAL" using  hab,
                             QMSG-PARAM1,
                             ValuesCount,
                             mp1ValueTypes,
                             mp1ValueData,
                             dummy

        move mp1Values(1) to idControl

* If user has double clicked on a list box item, simulate ENTER
* pushbutton selection

        if (idControl = LN-ENTER)

* Set new value for mp1

          move CMDSRC-OTHER to mp1Values(1)
          move DID-OK       to mp1Values(2)

          call "WUSVAL" using  hab,
                               ValuesCount,
                               mp1ValueTypes,
                               mp1ValueData,
                               mp1Value,
                               dummy

          move 1 to mp2Value

          call "WMPOST" using  hwndListBox,
                               WM-COMMAND,
                               mp1Value,
                               mp2Value,
```

```
                                        dummy

* Else do nothing
       else
         continue

* End of switch on control type
       end-if

    else
* Else message is passed to default dialog procedure

       call "WCDDP" using  QMSG-HWND,
                           QMSG-MSGID,
                           QMSG-PARAM1,
                           QMSG-PARAM2,
                           dummy

* End of Switch on message type
    end-if.

 EndListBoxDlg.

*****************************************************************************
*                    End of Fourth Dialog Routine                           *
*****************************************************************************
```

MyListBoxDlgRoutine Walk Through

The list box dialog box displays a list of text strings, and allows a single selection from the list. When the message-processing loop in the main procedure encounters a message destination of hwndListBox, it calls the dialog routine MyListBoxDlgRoutine. The routine processes the WM-HELP, WM-PCONTROL, and WM-COMMAND messages.

The WM-HELP Message

The WM-HELP message is processed by calling the function HelpProc.

The WM-PCONTROL Message

When the user interacts with a list box it sends a WM-CONTROL message to its owner. The notification code tells the owner the nature of the interaction. Similarly to the entry field example, the routine is not affected by the user's interaction with the list box until the user decides to enter a choice to the program. When the user presses the Enter or Return key when the list box has the focus, or when the user double-clicks the list box, the control sends a WM-CONTROL message with a notification code of LN-ENTER to its owner, the dialog box. The message is intercepted by the language-support dialog procedure and posted on as a WM-PCONTROL. The dialog box routine calls WUQVAL (WinQueryValue) to extract the notification code from the WM-PCONTROL message. If the notification code is LN-ENTER, the routine calls WUSVAL (WinSetValue) to construct a WM-COMMAND message, which it then posts to itself using WMPOST (WinPostMsg). The values CMDSRC-OTHER and DID-OK in the message parameters simulate the user selecting the **Enter** pushbutton or pressing the Enter key.

The WM-COMMAND Message

When the user selects a pushbutton, its system-provided window procedure posts a WM-COMMAND message to its owner, the dialog box. The WM-COMMAND message also occurs as a result of the WM-PCONTROL message processing (see last section). The first parameter of the message contains the command value of the pushbutton, DID-OK for **Enter**, and DID-CANCEL for **Cancel**. The dialog routine calls WUQVAL (WinQueryValue) to extract the command value, and places it in idCommand.

DID-OK Command Value

The routine calls WCSDIM (WinSendDlgItemMsg) to send an LM-QUERYSELECTION message to the list box, to find out the selected item. The index of the item is returned in the least significant half of the variable Reply. The routine calls WUQVAL (WinQueryValue) to extract the value, and puts it in idItem.

When the item has an index of LIT-NONE, it means that no item is selected. The routine simply sets SelectionSize to zero, and processing drops through to the WIDEL call that destroys the dialog box.

When the index indicates that an item is selected, the routine sends an LM-QUERYITEMTEXT message to the list box to obtain the text of the selected item. The routine calls WUSVAL (WinSetValue) to construct the first parameter of the message. The second parameter of the message is a handle to the structure in which you want the queried text to be returned. The WUCDS (WinCreateDataStructure) call creates a data structure and returns the structure handle SelectionHandle. Unlike the processing described in "The MI-LISTBOX Command Value" on page 12-13, where the WUCDS call created a data structure and initialized it with data, this time the call creates a data structure that is to receive data. Therefore, SelectionSize specifies that the initialization data has a zero length.

The WCSDIM (WinSendDlgItemMsg) call sends an LM-QUERYITEMTEXT message to the list box and places the queried string in the structure identified by SelectionHandle. The WUQDS (WinQueryDataStructure) call queries the data structure, placing the contents of the structure in Selection, which is the program form of the structure used in the routine ClientPaint. The WCDDS (WinDestroyDataStructure) call destroys the data structure.

The WIDEL (WinDestroyWindow) call destroys the dialog box, and a call to WINVRG (WinInvalidateRegion) sends a WM-PAINT to the client window to tell it to repaint itself.

DID-CANCEL Command Value

If the command value is DID-CANCEL, the routine calls WIDEL (WinDestroyWindow) for the dialog box, and WINVRG (WinInvalidateRegion) for the client window, and sets the dialog box handle to zero.

Default Dialog Processing

The program passes unprocessed messages to WCDDP (WinDefDlgProc), the default dialog procedure.

Using Data Structures

In C/2, a structure that is a parameter to an API call is usually passed as a pointer. For most calls in COBOL and FORTRAN that have structures as parameters (for example, WIQPOS (WinQueryWindowPos)), the structure itself is passed.

In COBOL and FORTRAN, some of the calls use a handle of data type HSTRUCT as a parameter. When you want to use such a call, you must first use the WUCDS (WinCreateDataStructure) call to create the structure and return the handle for use in your program.

"The MI-LISTBOX Command Value" on page 12-13 describes how the DLG2COB sample program (and its FORTRAN equivalent) uses WUCDS (WinCreateDataStructure) to create a simple data structure, and gets the structure handle so that data can be passed to PM. A more complex example is described in "Creating Structures" on page 12-36.

"DID-OK Command Value" describes how the samples use WUCDS (WinCreateDataStructure) to create a data structure and get a structure handle so that data can be *queried* from PM. In this case, the call WUQDS (WinQueryDataStructure) is used to move the data from the structure to your program storage.

In both cases, the returned handle is used as an input parameter to the call WCSDIM (WinSendDlgItemMsg).

Structures in Structures

In C/2, some structures contain pointers to other structures. In COBOL and FORTRAN, instead of using a pointer to a structure, you use a handle of type HSTRUCT. PM provides the call WUCDS (WinCreateDataStructure) to create a data structure. This call returns the handle of the data structure to your program, and your program must provide the storage to contain the handle.

When you need to change data in a structure, you use the WUMDS (WinModifyDataStructure) call.

Creating Structures

The following example shows how to create the DRIVDATA structure. This structure is used by presentation drivers to hold information about the output devices it supports. Its use is described in Chapter 37, "Creating a Printer Device Context."

```
*
*   Define the parameters for WUCDS.
*
*
*   Number of elements in the DRIV-TYPES array
 77 DRIV-COUNT              PIC S9(9) USAGE BINARY VALUE 10.
*
*   Data type codes that define the layout of the input
*   structure DRIV-DATA
 01 DRIV-TYPES.
    03  DRIV-TYPE           PIC S9(9) USAGE BINARY OCCURS 10.
*
*   Length (in bytes) of the input structure, DRIV-DATA
 77 DRIV-INLEN              PIC S9(9) USAGE BINARY VALUE 61.
*
*   The input data structure
*
 01 DRIV-DATA.
*   Length of the input data structure
    03 DRIV-LENGTH          PIC S9(9) USAGE BINARY VALUE 61.
*   Version number of the data
    03 DRIV-VERSION         PIC S9(9) USAGE BINARY VALUE 99.
*   Presentation-driver name information
    03 DRIV-DEV-NAME.
*      Length of driver name
       05 DRIV-LENGTH-2     PIC S9(9) USAGE BINARY VALUE 18.
*      Driver name
       05 DRIV-DEVICENAME   PIC X(32) VALUE "DRIVER DEVICE NAME".
*   Driver-dependent data
    03 DRIV-GENERALDATA.
*   The following fields are examples of driver data
       05 DRIV-X            PIC X VALUE "X".
       05 DRIV-ARRAY.
          07 DRIV-N1        PIC S9(9) USAGE BINARY VALUE 111.
          07 DRIV-N2        PIC S9(9) USAGE BINARY VALUE 222.
          07 DRIV-N3        PIC S9(9) USAGE BINARY VALUE 333.
          07 DRIV-N4        PIC S9(9) USAGE BINARY VALUE 444.
       05 FILLER REDEFINES DRIV-ARRAY.
          07 DRIV-N         PIC S9(9) USAGE BINARY OCCURS 4.
*
*   Maximum length of the standard form of the data structure.
*   Set to 0 (PM allocates sufficient storage).
 77 DRIV-MAXLEN              PIC S9(9) USAGE BINARY VALUE 0.
*
*   Handle of the data structure that is created
 77 DRIV-HANDLE              PIC S9(9) USAGE BINARY VALUE 0.
```

```
      :
*
*   Initialize the data types array
*
*   DRIV-COUNT...
     MOVE DTYP-CTL-LENGTH TO DRIV-TYPE(1).
     MOVE DTYP-LONG TO DRIV-TYPE(2).
     MOVE -1 TO DRIV-TYPE(3).
     MOVE -1 TO DRIV-TYPE(4).
*   DRIV-VERSION...
     MOVE DTYP-LONG TO DRIV-TYPE(5).
*   DRIV-DEVICENAME...
     MOVE DTYP-STR32 TO DRIV-TYPE(6).
*   DRIV-GENERALDATA (DRIV-X)...
     MOVE DTYP-BYTE TO DRIV-TYPE(7).
*   DRIV-GENERALDATA (DRIV-ARRAY)...
     MOVE DTYP-CTL-ARRAY TO DRIV-TYPE(8).
     MOVE DTYP-LONG TO DRIV-TYPE(9).
     MOVE -4 TO DRIV-TYPE(10).
*
*   Call WUCDS to create the data structure
*
    CALL "WUCDS" USING HAB
                       DRIV-COUNT
                       DRIV-TYPES
                       DRIV-INLEN
                       DRIV-DATA
                       DRIV-MAXLEN
                       DRIV-HANDLE
                       RESULT.
```

The example code first initializes the DRIV-TYPES array. This array defines the layout of the structure you want to create, in terms of its standard form[1].

The number of elements in the array is defined by the DRIV-COUNT field. The values you allocate to the elements of the array are defined in the WinRegisterUserDatatype (WMRUD) call. The required DTYP_ values are predefined in the constants files, PM.CIN and PM.FIN. One or more entries are required in the DRIV-TYPES array for each field of the standard form[1] of the structure.

The DRIV-VERSION field is a simple field, so a single entry of DTYP-LONG is sufficient.

The contents of the DRIV-GENERALDATA field are variable because they are driver-dependent. For this reason, the DRIV-LENGTH field needs four entries in the DRIV-TYPES array:

1. The control-data type, DTYP-CTL-LENGTH.
2. The data type of the field that contains the length, DTYP-LONG.
3. The index within DRIV-TYPES of the first structure component included in the length.
4. The index within DRIV-TYPES of the last structure component included in the length.

In the example, entries 3 and 4 are set to −1. This means that they refer to the start and to the end of the structure, respectively.

[1] The *standard* form of a structure is the form that is described in the *Control Program Programming Reference.*

The DRIV-GENERALDATA field contains a single byte followed by a fixed-length array, DRIV-ARRAY. It needs three entries in DRIV-TYPES to define the array:

1. The control-data type, DTYP-CTL-ARRAY.
2. The data type of the array elements, DTYP-LONG.
3. The count of the number of elements in the array. The count is negative, which implies that the array is of fixed length.

The WUCDS call has the following parameters:

1. DRIV-COUNT, the number of elements in the array.
2. The DRIV-TYPES array, which defines the layout of the required structure.
3. DRIV-INLEN, the length of the COBOL input structure (DRIV-DATA).
4. DRIV-DATA, the COBOL input structure itself.
5. DRIV-MAXLEN, the length of the standard form of the structure. DRIV-MAXLEN has a value of 0 in the example, so sufficient storage is allocated by PM to hold the standard form of the structure.
6. DRIV-HANDLE, the handle of the structure created.

Destroying Structures

In all cases, when your program has finished using a structure, it must call WUDDS (WinDestroyDataStructure) to destroy the structure and free the storage allocated by PM. Failure to do so can result in storage errors.

You must also do this for structure handles that are aliases returned by PM. For more information, see the description of the WinQueryDataStructure call in the *Programming Reference: Volume 1*.

Summary of Structure-Handling Calls

WUCDS (WinCreateDataStructure)
Creates a data structure in a standard form, and returns a handle to it.

WUDDS (WinDestroyDataStructure)
Destroys a data structure.

WUMDS (WinModifyDataStructure)
Modifies a data structure stored in a standard form.

WUQDS (WinQueryDataStructure)
Decomposes a data structure held in a standard form into the form required by your program.

Chapter 13. Additional Windowing Techniques

The preceding chapters of this book use the sample applications HELLO1, HELLO2, DIALOG1, and DIALOG2 to introduce the concepts and fundamentals of programming with windows. This chapter describes additional windowing techniques and considerations, some of which are mentioned in previous chapters, and illustrates them, where possible, with code from other sample programs supplied with the programming tools.

The subjects covered in this chapter are:

- Limits imposed by Presentation Manager
- Using presentation parameters
- Defining internal messages for your application
- Registering a public window class
- Registering a private window class
- Subclassing a window
- Creating an object window
- Using window words
- Controlling the state of the keyboard
- Hooks
- The atom manager
- The heap manager.

Presentation Manager Limits

Within Presentation Manager there are limits that cannot be exceeded. When one of these limits is reached during the running of an application, the system warns the application. This section tells you what the limits are and gives advice on how to handle them.

Table 13-1. Presentation Manager Limits

Parameter	Limit
Max. windows	1200
Max. string atoms	16384
Max. integer atoms	49152
Max. heap size, one segment	64KB[1]
Max. segments per presentation space	16378
Max. logical fonts per presentation space	254
Max. size logical color table	31KB[1]
Max. size per segment of presentation space	64KB[1]
Max. presentation spaces per process	32
Max. horizontal scan lines per cursor	32
Max. cursor width in text mode	1

[1] 1KB = 1024 bytes.

Recognizing and Recovering from Limits

When an application issues a call that would cause a limit to be exceeded, a return code is set to indicate the problem. The return codes for each call are given in the *Programming Reference: Volume 1*.

Recovery from limits is dependent on the application and the context in which the limit was reached. Here are some suggestions for handling common occurrences:

Limit Reached by User Action

When a limit is reached as a result of a user-requested action:

1. Display a message box to tell the user that a limit has been reached. If possible, suggest a method of solving the problem.
2. Restore your application to the condition it was in before the limit was reached.

Limit Reached During Initialization

When a limit is reached during application initialization:

1. Display a message box to tell the user that a limit has been reached. If possible, suggest a method of solving the problem.
2. Terminate the initialization.

Limit Reached During a Long Operation

When a limit is reached during a long operation, previous conditions often cannot be restored, so:

1. Try to bypass the limit in the application. If user assistance is required, display a message box to tell the user of the problem and explain the action required to resolve it.
2. Give the user the option of terminating the application.

Using Presentation Parameters

The presentation parameters of a window contain attributes about its visual appearance. The presentation parameters are held in the PRESPARAMS structure, which itself contains the PARAM structure. PRESPARAMS is part of the CREATESTRUCT and WNDPARAMS structures.

The elements of PRESPARAMS are:

attrcount
The number of presentation parameter type/value pairs

param
An array of attribute parameters. This is in the form of a PARAM structure.

The elements of PARAM are:

attrtype
A unique ID for the presentation parameter type. It is possible to create private presentation parameters. To ensure that the IDs of such private parameters do not conflict with the public ones, define them using the Atom Manager API (see "The Atom Manager" on page 13-21).

attrvaluelen
A byte count of the **attrvalue** parameter.

attrvalue
Value to which the type is being set.

You can have several PARAM structures within the PRESPARAMS structure.

The presentation parameter types are:

PP_FOREGROUNDCOLOR
RGB definition of foreground color

PP_BACKGROUNDCOLOR
RGB definition of background color

PP_FOREGROUNDCOLORINDEX
Index definition of foreground color

PP_BACKGROUNDCOLORINDEX
Index definition of background color

PP_HILITEFOREGROUNDCOLOR
RGB definition of highlighted foreground color

PP_HILITEBACKGROUNDCOLOR
RGB definition of highlighted background color

PP_HILITEFOREGROUNDCOLORINDEX
Index definition of highlighted foreground color

PP_HILITEBACKGROUNDCOLORINDEX
Index definition of highlighted background color

PP_DISABLEDFOREGROUNDCOLOR
RGB definition of disabled control foreground color

PP_DISABLEDBACKGROUNDCOLOR
RGB definition of disabled control background color

PP_DISABLEDFOREGROUNDCOLORINDEX
Index definition of disabled control foreground color

PP_DISABLEDBACKGROUNDCOLORINDEX
Index definition of disabled control background color

PP_BORDERCOLOR
RGB definition of control border color

PP_BORDERCOLORINDEX
Index definition of control border color

PP_FONTNAMESIZE
Font type and size.

The values for the PP_HILITE type are:

PPH_NORMAL
Normal.

PPH_UNREADABLE
Invisible. An example of the use of this attribute is to prevent typed characters being displayed in the entry field for a password.

PPH_BLINK
Blinking.

PPH_REVVIDEO
Reverse video.

PPH_UNDERLINE
Underlined.

PPH_BRIGHT
Bright.

The system-defined structure definitions of PRESPARAMS and PARAM are:

```
typedef struct _PRESPARAMS {
   ULONG cb;        /* count of type/value pairs */
   PARAM aparam[1]; /* array of parameters       */
} PRESPARAMS;

typedef struct _PARAM {
   ULONG id;        /* ID of parameter type      */
   ULONG cb;        /* count of attribute values */
   BYTE  ab[1];     /* attribute value           */
} PARAM;
```

Presentation Parameter Function Calls

Presentation Manager provides calls with which an application can set, query, and remove presentation parameters.

WinSetPresParam
Sets presentation parameters for a window

WinQueryPresParam
Retrieves information on presentation parameters

WinRemovePresParam
Removes a presentation parameter from a window.

The WM_PRESPARAMCHANGED message is sent when the presentation parameters for a window are changed by any of the above calls. The message is sent to all windows owned by the owner of the changed presentation parameter. This allows owned windows to inherit the change.

Defining Your Own Messages

Presentation Manager provides three classes of message:

System-defined messages
Available for use by all Presentation Manager applications

Application-defined internal messages
Intended for use only by the application that defined them

Application-defined external messages
Intended for use across several applications.

Each message used by an application must have a unique identification. To achieve this, the following rules for message IDs must be followed:

1. System message identifiers are defined by Presentation Manager and all have a value less than WM_USER (=0x1000). The data types of the corresponding message parameters are also defined by the system.
2. Internal application-defined message identifiers, together with any special data types that they use, are defined by the application. Their values are defined by adding a zero-based offset to WM_USER. The resulting identifier must be less than WM_ATOMFIRST (=0xC000).
3. External application-defined messages, together with any special data types that they use, are defined by the application. Their identifiers are defined as a unique string. At run time, a value is allocated to each identifier by running the System Atom Manager. This guarantees that a unique ID is allocated for each message. The values are in the range WM_ATOMFIRST (=0xC000) to WM_ATOMLAST (=0xF000).

The GRAPHIC1 sample program shows the definition of a range of internal application messages:

```
#define STOP_THREAD    WM_USER+1
#define DRAW_ORDERS    WM_USER+2
#define VSCROLL        WM_USER+3
#define SIZING         WM_USER+4
#define ZOOM_IN        WM_USER+5
#define ZOOM_OUT       WM_USER+6
#define DUMMY_COMMAND  WM_USER+7
#define LEFTDOWN       WM_USER+8
#define MOUSEMOVE      WM_USER+9
#define LEFTUP         WM_USER+10
#define PRINT_SCREEN   WM_USER+11
#define FLUSH_COMMAND  WM_USER+12
```

The client window procedure of GRAPHIC1 posts most of the above messages to its asynchronous drawing thread, as a result of receiving system-defined messages. For example, a VSCROLL message is posted when WM_VSCROLL is received. A procedure in the asynchronous thread deals with the application-defined messages, so that the thread containing the client window procedure remains responsive to the user.

Registering Data Types and Messages

An application can register messages including new data types so that they are available for use by other applications. Application messages are defined and registered using WinRegisterUserMessage (WMRUM). The definition includes all the message parameters. This can include system- and application-defined data types.

Application data types are defined and registered using WinRegisterUserDatatype (WMRUD). When a data type is registered, a corresponding pointer data type is also registered. This has the same name, but prefixed with "P."

Example Code: Using External Application-Defined Messages

This example illustrates how to use messages that are defined in the system atom table. These messages are defined by an application for use by other applications. The window or dialog procedures that process such messages must therefore be compiled into DLLs so that the code can be called by other applications. A DLL typically requires two levels of initialization:

1. Initialization at DLL load time. The message atoms are static variables that must be initialized at load time.
2. Per-process initialization. The registration of window classes, user messages, and user data types takes place on a per-process basis.

DLL File Initialization

The following DLL-initialization routine must be explicitly called by each process before using window classes and dialogs in the DLL.

```
/*********************************************************************/
/*   OS/2 Headers                                                    */
/*********************************************************************/
#include <os2.h>

/*********************************************************************/
/*   Headers for this DLL                                            */
/*********************************************************************/
#include "mydll.h"

/*********************************************************************/
/*   Dynalink Static Data                                            */
/*********************************************************************/
static ULONG    semaphore=0;   /* semaphore to protect static data */
static HSEM     hsem=&semaphore;
static BOOL     initialised=FALSE; /* is static data initialized?  */
static USHORT   MY_MESSAGE1=0;     /* message atoms                */
static USHORT   MY_MESSAGE2=0;

/*********************************************************************/
/* Initialization Procedure                                          */
/*********************************************************************/
BOOL FAR PASCAL MyDllInit(hab)
HAB  hab;
{
   BOOL    reply;

/*********************************************************************/
/* Dynalink Load-time Initialization                                 */
/*********************************************************************/
   DosSemWait( hsem, (ULONG)(-1) );
   if ( !initialised )
   {
     MY_MESSAGE1 = WinAddAtom( WinQuerySystemAtomTable(),
                               "MY_MESSAGE1" );
     MY_MESSAGE2 = WinAddAtom( WinQuerySystemAtomTable(),
                               "MY_MESSAGE2" );
     initialised = (BOOL)MY_MESSAGE1 && (BOOL)MY_MESSAGE2;
```

```
    }
    reply = initialised;

/*********************************************************************/
/* Clear Initialization Semaphore                                    */
/*********************************************************************/
    DosSemClear(hsem);

/*********************************************************************/
/* Per-Process Initialization                                        */
/*********************************************************************/
    if ( reply )
    {
      reply = WinRegisterClass(hab,/* Register class                 */
                               /* for this process                  */
                  MYCLASSNAME,    /* string defined in mydll.h      */
                  MyClassWinProc,
                  MYCLASSSTYLE,   /* CS_* flags defined in mydll.h */
                  MYCLASSWORDS);  /* number of window words         */
                               /* words defined in mydll.h          */

      reply = WinRegisterUserMessage(hab, /* Register user messages */
                  MY_MESSAGE1,          /* this process             */
                  DTYP_MPARAM, RUM_IN,
                  DTYP_MPARAM, RUM_IN,
                  DTYP_MRESULT) && reply;
      reply = WinRegisterUserMessage(hab,
                  MY_MESSAGE2,
                  DTYP_MPARAM, RUM_IN,
                  DTYP_MPARAM, RUM_IN,
                  DTYP_MRESULT) && reply;
    }

/*********************************************************************/
/* Return to caller                                                  */
/*********************************************************************/
    return reply;
}
```

Note: The segment that contains the static variables MY_MESSAGE1 and MY_MESSAGE2 must not be marked DISCARDABLE. OS/2 must swap the segment into SWAPPER.DAT to preserve the atom values.

Window or Dialog Procedure

The C language CASE statement requires a compile time constant. An IF statement must be used for message atoms. Other than this, the window or dialog procedure that processes message atoms is the same as any other window or dialog procedure.

```
/*********************************************************************/
/*  Window Procedure                                                 */
/*********************************************************************/
MRESULT FAR PASCAL MyClassWinProc(hwnd, msgid, param1, param2 )
HWND     hwnd;
USHORT   msgid;
MPARAM   param1;
```

```
MPARAM   param2;
{
    MRESULT reply=0L;

    if ( msgid < WM_ATOMFIRST )
    {
/*********************************************************************/
/* #define messages                                                  */
/*********************************************************************/
      switch ( msgid )
      {
        case WM_CREATE:
          reply = MyCreateProc(hwnd, param1, param2);
          break;

        case WM_DESTROY:
          reply = MyDestroyProc(hwnd, param1, param2);
          break;

        case WM_PAINT:
          reply = MyPaint(hwnd);
          break;

        default:
           reply = WinDefWindowProc(hwnd, msgid, param1, param2);
         break;
      }
    }
/*********************************************************************/
/* Atom Messages                                                     */
/*********************************************************************/
    else if ( msgid == MY_MESSAGE1 )
      reply = MyMessage1(hwnd, param1, param2);
    else if ( msgid == MY_MESSAGE2 )
      reply = MyMessage2(hwnd, param1, param2);

    return( reply );

}
```

Calling Routine

The calling routine needs to call the initialization routine, MyDllInit, which registers classes and user messages as required. It also needs to initialize its own copy of the message atoms.

```
/*********************************************************************/
/*  OS/2 Headers                                                     */
/*********************************************************************/
#include <os2.h>
/*********************************************************************/
/*  Headers for the DLL                                              */
/*********************************************************************/
#include "mydll.h"
/*********************************************************************/
/*  Static Data                                                      */
```

```
/*********************************************************************/
static USHORT   MY_MESSAGE1=0;                  /* message atoms     */
static USHORT   MY_MESSAGE2=0;
HAB           hab =  NULL;
HMQ           hmq =  NULL;
HWND          hwnd = NULL;

/*********************************************************************/
/* Main Procedure                                                    */
/*********************************************************************/
int cdecl main(argc,argv)
int  argc;
char *lbrk.argv];
{

/*********************************************************************/
/* Initialize Presentation Manager                                   */
/*********************************************************************/
  if ( hab = WinInitialize(NULL) )
    if ( hmq = WinCreateMsgQueue(hab, 0) )
/*********************************************************************/
/* Initialize Dynalink                                               */
/*********************************************************************/
      if ( MyDllInit(hab) )
      {
        MY_MESSAGE1 = WinFindAtom( WinQuerySystemAtomTable(),
                                   "MY_MESSAGE1" );
        MY_MESSAGE2 = WinFindAtom( WinQuerySystemAtomTable(),
                                   "MY_MESSAGE2" );
      }

/*********************************************************************/
/* Create window and post message                                    */
/*********************************************************************/
      hwnd = WinCreateWindow( HWND_DESKTOP,
                              MYCLASSNAME,
                              "",
                              0L,
                              0,0,0,0,
                              NULL,
                              HWND_TOP,
                              ID,
                              NULL,
                              NULL);
      WinPostMsg(hwnd, MY_MESSAGE1, 0L, 0L);

/*********************************************************************/
/* Service message loop                                              */
/*********************************************************************/
      while( WinGetMsg(hab, (PQMSG)&qmsg, NULL, 0, 0 ) )
             WinDispatchMsg(hab, (PQMSG)&qmsg );
```

Note: WinFindAtom is used rather than WinAddAtom. The atoms must be in the system atom table because MyDllInit has returned successfully.

Registering a Public Window Class

A public window class is registered in a DLL that also contains its window procedure. For the DLL to be available to all the applications in the system, its name must be in a list of DLLs in OS2.INI. The user shell automatically loads and initializes the DLLs in the list at system initialization time. The relevant section of OS2.INI has the form:

Application name
SYS_DLLS

Key name
Load

Load list data string
A list of fully qualified dynamic link library modules, including a file-name extension (normally .DLL), or the name of a DLL file.

A DLL-installing application is required to add the DLL to OS2.INI. It can use WinWriteProfileString (WMWPST) to do this. However, before it adds the DLL to the list, it should use WinQueryProfileString (WMQPST) to check whether there are already any DLLs on the list. If there are, the new DLL(s) should be inserted at the end of the load list.

At system initialization time, each DLL is loaded in the load list order, and its entry point number 1 called. The DLLs are called under the process of the shell.

All DLLs in the list are automatically attached to a process when it issues its first WinCreateMsgQueue (WMCMQ) call. All of the OS/2-supplied standard window classes are registered before the DLLs are loaded.

A load-list DLL can register public window classes and issue non-PM calls. It cannot create queues or windows, or communicate with the user. It can register a public window class by specifying a class style of CS_PUBLIC with WinRegisterClass (WMRCL). *It is an error if the CS_PUBLIC class style is specified on any process other than the shell process.* It can be issued only from a DLL listed in the load list.

You can register a public class with the same name as an existing public class (either an OS/2-supplied one or one already registered from the load list). This has the effect of replacing the information provided by the earlier registration. If the replacing class wants to pass some of the messages on to the original class, the replacing class must do the following:

1. Issue WinQueryClassInfo (WMQCLI), and save the original window procedure address.
2. Issue WinRegisterClass (WMRCL), specifying the same class style, a count of additional window bytes not less than the original, and the new window procedure address.
3. The new window procedure must pass unprocessed messages on to the saved window procedure address.

Registering a Private Window Class

When an application registers a private window class with the window procedure in a DLL, it is the responsibility of the application to resolve the address of the window procedure before issuing the WinRegisterClass (WMRCL) call. It should do this (explicitly or implicitly) by issuing the DosLoadModule call. Alternatively, an application can use the WinLoadLibrary (WMLLIB) and WinLoadProcedure (WMLPR) calls.

It is an error for a process to register a private class with the same name as a public class or a private class in the same process.

Subclassing a Window

If there is a public or private window class whose window procedure provides most of the processing you want for a window, but either lacks some processing that you want performed, or contains processing that you do not want performed, or contains processing that you want to affect, there is a technique for intercepting messages input to the window procedure for the window class. You pass the intercepted messages to an application window procedure for processing *before* the other window procedure. Your application either provides the additional processing you require, or, when the original procedure contained processing that you do not want performed, your application procedure discards the relevant messages. Alternatively, you can change the content of the message, and pass it to the original procedure. The technique is called *subclassing*.

Note: Subclassing is not supported in COBOL/2 and FORTRAN/2.

The result of subclassing is a new window or window class that does some or all of the processing of the original window class, *and* new processing, if required.

To subclass a window, write a window procedure that processes the messages that you want to intercept and provides the additional functions that the original window procedure lacked. At the end of your new window procedure the original window procedure for the class is called instead of the default window procedure. Your new window procedure determines which messages it processes, which it discards, and which it passes on to the original window procedure.

You can subclass either an individual instance of a window that you have already created, or an entire window class.

To subclass an individual window, call WinSubclassWindow (WISUBW), using the window handle of the window and the address of the new window procedure as input parameters. You can subsequently reverse the effect of the call, if you want to, by calling WinSubclassWindow again with the address of the original window procedure.

To subclass an entire window class, you use WinRegisterClass (WMRCL) to register the address of the new window procedure for your new class name.

If you want to be able to create instances of the subclassed window, without subclassing the entire class, you can register a new window class using WinRegisterClass (WMRCL) with the address of your new window procedure as an input parameter.

You can use the WinQueryClassInfo (WMQCLI) call to obtain information about public window classes.

Example Code: Subclassing an Instance of a Window

The IMAGE sample program subclasses an instance of the frame window, to intercept and process the SC_MAXIMIZE command value of the WM_SYSCOMMAND message, and the WM_QUERYTRACKINFO message. To make it easier for you to follow the logic, the following code fragments are in a different order from the order in which they appear in IMAGE.

```
/*********************************************************************/
/*          Private Window Procedures and Dialog Procedures          */
/*********************************************************************/
  :
MRESULT EXPENTRY  FrameSubProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2);

/*********************************************************************/
/*                    Global Variable Declarations                   */
/*********************************************************************/
  :
PFNWP    pfnwpFrame;                    /* main frame-window procedure */
  :
```

The pfnwpFrame procedure holds the address of the old frame-window procedure. FrameSubProc uses it in its processing.

```
/********************** Start of Main Procedure ***********************/
/* MAIN:                                                              */
/* The MAIN procedure obtains an anchor block handle, creates a       */
/* message queue, and creates the required windows. It contains the   */
/* main message-processing loop, which dispatches messages to the     */
/* window procedure WNDPROCA. Message-processing terminates on receipt */
```

```
/* of the WM_QUIT message.                                              */
/*********************************************************************/

VOID cdecl main ( )
{
 QMSG  qmsgImg;

 habImage = WinInitialize( (USHORT)NULL );
 hmqImage = WinCreateMsgQueue( habImage, (SHORT)NULL );

 if( hmqImage != (HMQ)NULL )
     {
     if( WndCreate() )
        {

        while ( WinGetMsg( habImage, &qmsgImg, (HWND)NULL, 0, 0 ))
           WinDispatchMsg( habImage, &qmsgImg );

        /*****************************************************************/
        /* Ensure that the scrollbars are child windows of the frame,  */
        /* so that they will be destroyed automatically.               */
        /*****************************************************************/
        if( !WinIsChild( hwndHscr, hwndFramei ))
            {
            WinSetParent( hwndVscr, hwndFramei, FALSE );
            WinSetParent( hwndHscr, hwndFramei, FALSE );
            }
        WinDestroyWindow( hwndFramei );
        }
     TermSelectFileDlgRes();
     WinDestroyMsgQueue( hmqImage );
     }
  WinTerminate( habImage );
  DosExit( EXIT_PROCESS, 0 );
}

/********************** End of Main Procedure ************************/

  :
/***************** Start of WNDCREATE Private Function *****************/
/* WNDCREATE:                                                        */
/* This function registers separate classes for the image window and   */
/* the frame subclass procedure, and creates a standard window.        */
/* WNDCREATE is invoked from the main procedure.                       */
/*********************************************************************/

BOOL WndCreate()
{
 ULONG  flCreate;

  /*******************************************************************/
  /* Separate classes are registered for the image window and the     */
  /* frame subclass procedure.                                       */
  /*******************************************************************/

  if( !WinRegisterClass( habImage
                       , (PSZ)"IMAGE"
                       , (PFNWP)WndProcA
                       , 0L
                       , 0 ))
```

```
    return( FALSE );


  if( !WinRegisterClass( habImage
                       , (PSZ)"SUBFRAME"
                       , (PFNWP)FrameSubProc
                       , 0L
                       , 0 ))
    return( FALSE );

  /*********************************************************************/
  /* A standard window with standard controls is created.              */
  /*********************************************************************/

  flCreate = FCF_STANDARD | FCF_VERTSCROLL | FCF_HORZSCROLL;

  hwndFramei = WinCreateStdWindow(
                    (HWND)HWND_DESKTOP
                  , WS_VISIBLE
                  , &flCreate
                  , (PSZ)"IMAGE"
                  , (PSZ)NULL
                  , WS_CLIPCHILDREN
                  , (HMODULE)NULL
                  , IDM_IMAGE
                  , &hwndImage ); /* client window handle  */

   WinQueryWindowText( hwndFramei, sizeof(szMBtitle), szMBtitle );


  :

  /*********************************************************************/
  /* The frame window procedure is subclassed, so that frame-sizing    */
  /* restrictions can be implemented.                                  */
  /*********************************************************************/
   pfnwpFrame = WinSubclassWindow( hwndFramei, (PFNWP)FrameSubProc );

  :
  /*********************************************************************/
  /* Allocate storage required by File Select dialog                   */
  /*********************************************************************/
  pszFile = InitSelectFileDlgRes(&habImage,
                                 &hwndFramei,
                                 (PSZ)szMBtitle,
                                 IDS_DEFMASK);
  if( pszFile == (PSZ)NULL)
     return(FALSE);
  else
     return(TRUE);
}

/****************** End of WNDCREATE Private Function *****************/
  :
/*************Start of Frame-Window Subclass Procedure ****************/
/* FRAMESUBPROC:                                                     */
/* The purpose of the frame-window subclass procedure is to restrict */
/* frame-window sizing, so that it is in step with the size of the   */
/* client-window presentation space. That is, the size of the image  */
```

```
/* dictates the size of the maximized window. Messages intended for    */
/* the frame-window procedure are sent here first.                      */
/************************************************************************/

MRESULT EXPENTRY  FrameSubProc( hwnd, msg, mp1, mp2 )
HWND    hwnd;
USHORT  msg;
MPARAM  mp1;
MPARAM  mp2;
{
  PTRACKINFO ptrack;
  RECTL      rcl;
  POINTL     ptl;

    switch( msg )
        {
        case WM_QUERYTRACKINFO:

           /*********************************************************/
           /* Invoke the normal frame-window procedure first, in order */
           /* to update the tracking rectangle to the new position.    */
           /*********************************************************/

           (*pfnwpFrame)( hwnd, msg, mp1, mp2 );
           ptrack = (PTRACKINFO)mp2;

           if( fImgLoaded && fBrowse )
             {

             /*******************************************************/
             /* Limit the size of the bounding rectangle only if the   */
             /* window is being sized.                                 */
             /* fs and rclBoundary are tracking information structure  */
             /* elements.                                              */
             /*******************************************************/

             if((( ptrack->fs & TF_MOVE  ) != TF_MOVE ) &&
                (( ptrack->fs & TF_LEFT  ) ||
                (( ptrack->fs & TF_TOP   ) ||
                (( ptrack->fs & TF_RIGHT ) ||
                (( ptrack->fs & TF_BOTTOM ) ||
                (( ptrack->fs & TF_SETPOINTERPOS )))
                {
                WinQueryWindowRect( hwndImage, &rcl );
                GpiQueryCurrentPosition( hpsImage, &ptl );

                track->rclBoundary.yTop    = rcl.yBottom + ptl.y;
                track->rclBoundary.yBottom = rcl.yBottom + ptl.y - sizlImg.cy;

                track->rclBoundary.xLeft   = rcl.xLeft + ptl.x;
                track->rclBoundary.xRight  = rcl.xLeft + ptl.x + sizlImg.cx;

                /*****************************************************/
                /* Convert client boundary coordinates to screen       */
                /* co-ordinates.                                       */
                /*****************************************************/

                WinMapWindowPoints( hwndImage,
                                    HWND_DESKTOP,
                                    (PPOINTL)&ptrack->rclBoundary,
```

```
                                        2 );

                    /*****************************************************/
                    /* Calculate equivalent frame boundary from client   */
                    /* boundary data.                                    */
                    /*****************************************************/

                    WinCalcFrameRect( hwndFramei,
                                      (PRECTL)&ptrack->rclBoundary,
                                      FALSE );

                    ptrack -> fs |= TF_ALLINBOUNDARY;
                    }
                  return(MRESULT)( TRUE );
                  }
                else
                  return(MRESULT)( TRUE );
                break;

            default:

                /*********************************************************/
                /* Return via the normal frame window proc.              */
                /*********************************************************/

                return( (*pfnwpFrame)( hwnd, msg, mp1, mp2 ));
                break;

            }
        return( (MRESULT)NULL );
    }
    /*************** End of Frame-Window Subclass Procedure ***************/
```

Creating an Object Window

An object window is one that is not displayed, but is created purely to process messages and do work in a similar way to a subroutine. To create an object window, use WinCreateWindow (WICRT) with a parent window handle of either HWND_OBJECT, or an object window handle returned by WinQueryObjectWindow (WIQOBJ).

Using Window Words

There is an area of storage that you can allocate as a parameter to WinRegisterClass (WMRCL) and subsequently use in your application. The area of storage is called the window words and can be used by each window of the registered class.

System-provided window classes have system-defined window words that contain QWS_ and QWL_ values. The WC_FRAME, WC_LISTBOX, WC_BUTTON, WC_STATIC, WC_ENTRYFIELD, WC_SCROLLBAR, and WC_MENU classes also have a window word at offset QWL_USER that your application can use to pass application-specific data to the window classes.

For application-defined window classes, your application allocates the data for the window words, and you decide to what use your application puts it. The size of the windows word area is defined when the class is registered. Typically, it is not very large.

The code in "Example Code: Window Words" on page 13-15 is adapted from the DIALOG1 sample. It uses only a small amount of data, but shows how to use window words if a window needs large volumes of data. The example allocates a block of memory to hold the data and places the location of the block of memory in the window words area. It retrieves some data from an entry field and places it in the block of memory.

Example Code: Window Words

A code sample based on DIALOG1.C to demonstrate the use of Window Words.

```
#define INCL_DOSMEMMGR
#define INCL_WINSYS                     /* Selectively include          */
#define INCL_WINWINDOWMGR               /* the relevant parts           */
#define INCL_GPI                        /* of the Presentation          */
#include <os2.h>                        /* Manager header file.         */
#include <string.h>                     /* C/2 string functions         */
#include "dialog1.h"                    /* Resource symbolic identifiers*/

#define LENGTH_STRING  33               /* Length of string             */

MRESULT EXPENTRY MyWindowProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 );
MRESULT EXPENTRY MyDlgProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 );
VOID cdecl main( VOID );

typedef struct{ char szLocation-LENGTH_STRING];
             } MYWINDOWDATA;            /* Define a structure           */

                                        /* Define parameters by type    */
HAB   hab;                              /* Anchor block handle          */
HMQ   hmq;                              /* Message queue handle         */
HWND  hwndClient;                       /* Client window handle         */
ULONG flCreate = FCF_STANDARD;          /* Frame control flag           */
BOOL  bComplete = FALSE;                /* Switch for first time through*/
HWND  hwndFrame;                        /* Frame window handle          */
CHAR  szResponse-LENGTH_STRING] = "";   /* To hold "You live in".       */

VOID cdecl main(  )
{
  QMSG  qmsg;

  hab = WinInitialize( NULL );
  hmq = WinCreateMsgQueue( hab, 0 );

  WinRegisterClass(                     /* Register Window Class        */
      hab,                              /* Anchor block handle          */
      "MyWindow",                       /* Window Class name            */
      MyWindowProc,                     /* Address of Window Procedure  */
      CS_SIZEREDRAW,                    /* Class Style                  */
      sizeof( ULONG )                   /* Reserve window words area    */
      );


  hwndFrame = WinCreateStdWindow(
        HWND_DESKTOP,                   /* Desktop Window is parent     */
        WS_VISIBLE,                     /* Frame Style                  */
        &flCreate,                      /* Frame Control Flag           */
        "MyWindow",                     /* Window Class name            */
        "",                             /* No window title              */
        0L,                             /* Client style of visible      */
        NULL,                           /* Resource is in .EXE file     */
        ID_MAINWND,                     /* Frame window identifier      */
        (PHWND)&hwndClient              /* Client window handle         */
        );
```

```
  while( WinGetMsg( hab, (PQMSG)&qmsg, (HWND)NULL, 0, 0 ) )
    WinDispatchMsg( hab, (PQMSG)&qmsg );

  WinDestroyWindow( hwndFrame );        /* Tidy up...                    */
  WinDestroyMsgQueue( hmq );            /* and                           */
  WinTerminate( hab );                  /* terminate the application     */
}

MRESULT EXPENTRY MyWindowProc ( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
  HPS    hps;                           /* Presentation space handle     */
  POINTL pt;                            /* String screen coordinates     */
  RECTL  rc;                            /* Window rectangle              */
  MYWINDOWDATA *p;                      /* Declare pointer to structure */


  switch( msg )
  {
    case WM_CREATE:
      /* Window initialization - load "You live in" string from resource*/
      WinLoadString( hab,
                     (HMODULE)NULL,        /* Resource is in .EXE file  */
                     ID_RESPONSE,          /* String identifier         */
                     sizeof( szResponse ), /* Size of buffer            */
                     (PSZ)szResponse       /* Buffer                    */
                   );
      OFFSETOF( p ) = 0;                   /* Set offset to zero        */
      DosAllocSeg( sizeof( MYWINDOWDATA ), (PSEL)&SELECTOROF( p ), 0 );
      WinSetWindowULong( hwnd, 0, (ULONG)p ); /* Store the pointer      */
      strcpy( p -> szLocation, "Initial" );   /* Initialize a field     */
      break;
    case WM_DESTROY:
      p = (MYWINDOWDATA *)WinQueryWindowULong( hwnd, 0 );/* Load pointer*/
      DosFreeSeg( SELECTOROF( p ) );                     /* Free storage*/
      break;

    case WM_COMMAND:
      switch ( SHORT1FROMMP( mp1 ) )
      {
        case ID_WHERE:
          WinDlgBox( HWND_DESKTOP,      /* Place anywhere on desktop    */
                     hwndFrame,         /* Owned by frame               */
                     MyDlgProc,         /* Address of dialog procedure  */
                     NULL,              /* Module handle                */
                     ID_MYDIALOG,       /* Dialog identifier in resource*/
                     NULL );            /* Initialization data          */
          WinInvalidateRegion( hwnd, NULL, FALSE ); /* Force a repaint  */
          break;
        case ID_EXITPROGRAM:
          WinPostMsg( hwnd, WM_CLOSE, 0L, 0L );
          break;
        default:
      return WinDefWindowProc( hwnd, msg, mp1, mp2 );
      }
      break;
    case WM_PAINT:
      hps = WinBeginPaint( hwnd, (HPS)NULL, (PRECTL)&rc );
      WinFillRect( hps, (PRECTL)&rc, SYSCLR_WINDOW );
      if( bComplete)
```

```
      {
        GpiSetColor( hps, SYSCLR_WINDOWTEXT ); /* Text color same as PM */
        pt.x = 50L; pt.y = 50L;          /* Coordinates of "You live in" */
        GpiCharStringAt( hps, &pt, (LONG)strlen( szResponse ), szResponse );
        p = (MYWINDOWDATA *)WinQueryWindowULong( hwnd, 0 );/* load p    */
        pt.x = 50L; pt.y = 30L;          /* Coordinates of location      */
        GpiCharStringAt( hps, &pt, (LONG)strlen( p -> szLocation ),
                         p -> szLocation );
      }
      WinEndPaint( hps );                /* Drawing is complete          */
      break;
    case WM_CLOSE:
      WinPostMsg( hwnd, WM_QUIT, 0L, 0L );      /* Cause termination    */
      break;
    default:
      return WinDefWindowProc( hwnd, msg, mp1, mp2 );
  }
  return FALSE;
}

MRESULT EXPENTRY MyDlgProc( HWND hwndDlg, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
  MYWINDOWDATA *p;

  p = (MYWINDOWDATA *)WinQueryWindowULong( hwndClient, 0 );
  switch ( msg )
  {
    case WM_COMMAND:                     /* Posted by pushbutton or key  */
      switch( SHORT1FROMMP( mp1 ) )      /* Extract the command value    */
      {
        case DID_OK:                     /* The Enter pushbutton or key. */
                                         /* WinQueryWindowText copies the*/
                                         /* contents of the entry field  */
                                         /* into szLocation.             */
          WinQueryWindowText( WinWindowFromID( hwndDlg, ID_ENTRYFIELD ),
                              LENGTH_STRING,
                              p -> szLocation );
          bComplete = TRUE;              /* Set switch so strings are    */
                                         /* drawn in WM_PAINT processing.*/
        case DID_CANCEL:         /* The Cancel pushbutton or Escape key */
          WinDismissDlg( hwndDlg, TRUE );  /* Removes the dialog box    */
          return FALSE;
        default:
          break;
      }
      break;
    default:
      return WinDefDlgProc( hwndDlg, msg, mp1, mp2 );
  }
  return FALSE;
}
```

The TYPEDEF statement defines the structure MYWINDOWDATA, which holds the single field, szLocation. This is the structure of the block of memory that holds the contents of the entry field.

In the MAIN() procedure, the last parameter of WinRegisterClass (WMRCL) reserves the number of window words that can be used by a window of this class. The C/2 operator SIZEOF allocates one window word only (the size of a ULONG).

In the MyWindowProc window procedure, a pointer (p) is declared that is stored in the window word and points to a block of memory that has a structure corresponding to MYWINDOWDATA.

In the processing of the WM_CREATE message, the offset part of the pointer is set to zero. DosAllocSeg allocates a block of memory that is the size of the MYWINDOWDATA structure, and returns the address of the memory in the selector portion of p. The final parameter of the call is an option flag that specifies the memory as nonsharable.

WinSetWindowULong (WISWUL) stores the pointer in the window word. The second parameter of the call is expressed in bytes, and holds a value from an index where the base value is zero, and the top value is four bytes less than the number of words reserved by WinRegisterClass (WMRCL). In the example, as WinRegisterClass reserved one word only, the index value is zero. If WinRegisterClass reserves more than one window word, the index value, which can also be used in the query call WinQueryWindowULong (WIQWUL), can be set to the index value that corresponds to the window word that you want to set or query.

The C/2 library function STRCPY copies the string "Initial" into the szLocation field pointed to by pointer p.

In the processing of WM_DESTROY, the value of the pointer is loaded from the window word, and is used to free the storage.

In the processing of WM_PAINT, WinQueryWindowULong is used to load the value in the window word into the pointer p. GpiCharStringAt (GICHAR) uses the p to point to the string in the szLocation in the MYWINDOWDATA structure, both to get the length of the string, and to get the string itself.

In the dialog procedure MyDlgProc, the pointer p is declared again because it is a local variable. The value of p is loaded from the window word. The WinQueryWindowText (WIQTXT) call copies the text that the user enters from the entry field into szLocation.

Window Words Function Calls

WinRegisterClass (WMRCL)
Registers a window class and allocates the number of extra window words for each window of a window class.

WinSetWindowULong (WISWUL)
Sets an unsigned long integer value into the reserved window word's memory.

WinSetWindowUShort (WISWUS)
Sets an unsigned short integer value into the reserved window word's memory.

WinQueryWindowULong (WIQWUL)
Gets the unsigned long integer value at a specified offset from the reserved window word's memory.

WinQueryWindowUShort (WIQWUS)
Gets the unsigned short integer value at a specified offset from the reserved window word's memory.

Controlling the State of the Keyboard

In a windowed environment, applications may need to control the state of the keyboard Num Lock, Caps Lock, and Scroll Lock. To allow the application keyboard state to be maintained, Presentation Manager updates its logical keyboard each time a change is made. Applications need to do the following to allow each application to use its own keyboard state:

1. Provide storage to remember its own logical keyboard state and the Presentation Manager logical keyboard state.
2. Save the Presentation Manager logical keyboard state when receiving the input focus, using the WinGetKeyState (WMKEYS) call.
3. Set the logical keyboard state when receiving the input focus, using the WinSetKeyboardStateTable (WMSKST) call.
4. Save all changes caused by WM_CHAR messages representing the application defined

CAPS LOCK, NUM LOCK, SCROLL LOCK and INSERT keys.

5. Issue a WinSetKeyboardStateTable call each time a WM_CHAR message is received containing VK_CAPITAL, VK_NUMLOCK, or VK_SCRLLOCK.
6. Call WinSetKeyboardStateTable to restore the Presentation Manager logical keyboard state when releasing the input focus.

Hooks

Hooks allow a Presentation Manager application to test for a specific event at various points in processing. They make the system call a hook procedure as a result of the event.

Note: Hooks are not supported in COBOL/2 and FORTRAN/2.

For a given event, you can link more than one hook procedure together into a *hook chain*. You chain the hook procedures together using successive WinSetHook (WMSHOO) calls to install each hook procedure at the head of the chain. The specified event is passed to each hook procedure in turn, in the reverse of the order in which the hook procedures were installed. (The most recently installed hook procedure is called first.) When a hook procedure returns FALSE, the next hook procedure in the chain is called. When a hook procedure returns TRUE, the next hook procedure is not called.

You can have a system hook chain or a queue hook chain:

- A **system** hook chain is a public hook chain that applies to any Presentation Manager application in the system. You can, therefore, install hooks that affect the entire system. A system hook chain must reside in a DLL.
- A **queue** hook chain is a private hook chain that applies only to the thread that creates the queue with which the hook chain is associated. It can, therefore, reside in the application, or in a DLL.

You use the WinSetHook (WMSHOO) call to install a hook. You specify the queue that is to be checked, the hook type, and the address of the hook procedure. Use the WinReleaseHook (WMRHK) call to remove a hook procedure.

The hook type has an HK_ prefix and it determines where in the processing the hook or hook chain operates:

HK_HELP
The hook is called by the default processing associated with any window in which the user requests help. The procedure can also be called directly by an application. See HK_HELP in the *Programming Reference: Volume 2.*

HK_INPUT
The hook is called when messages are removed from an application queue, before being returned by WinGetMsg (WMGET) or WinPeekMsg (WMPEEK). See HK_INPUT in the *Programming Reference: Volume 2.*

HK_JOURNALRECORD
The hook is called after message translation, but before keyboard input is translated by system or queue accelerator tables. The hook records user input messages. The message input is passed on to the application. See HK_JOURNALRECORD in the *Programming Reference: Volume 2.*

HK_JOURNALPLAYBACK
Plays back recorded message input. See HK_JOURNALPLAYBACK in the *Programming Reference: Volume 2.*

HK_MSGFILTER
The hook is called inside a system modal loop, according to an MSGF_ code input to the hook. Your application can also use WinCallMsgFilter (WMCMFI) to call the message filter hook directly. See WinCallMsgFilter in the *Programming Reference: Volume 1.* and HK_MSGFILTER in the *Programming Reference: Volume 2.*

HK_SENDMSG
The hook is called when WinSendMsg (WMSEND) sends a message. See HK_SENDMSG in the *Programming Reference: Volume 2.*

HK_CODEPAGECHANGE
This hook notifies that a message queue code page has been changed. It is sent after the new code page has been set. See HK_CODEPAGECHANGE in the *Programming Reference: Volume 2.*

Example Code: Using a Help Hook

As an example of how to use hooks, the following skeleton code shows how you use a help hook in an application.

```
  :
            /* Must be predeclared */

BOOL EXPENTRY HelpHook( HAB hab, SHORT sMode, SHORT idTopic,
                        SHORT idSubTopic, PRECTL prcPosition );
  :
            /* Start main procedure */
VOID cdecl main(  )
{
            /* Initialization */
  :

  WinSetHook( hab, HMQ_CURRENT, HK_HELP, (PFN)HelpHook, 0L );

  while( WinGetMsg( hab, (PQMSG)&qmsg, (HWND)NULL, 0, 0 ) )
    WinDispatchMsg( hab, (PQMSG)&qmsg );

  WinReleaseHook( hab, HMQ_CURRENT, HK_HELP, (PFN)HelpHook, 0L );
  WinDestroyWindow( hwndFrame );
  WinDestroyMsgQueue( hmq );
  WinTerminate( hab );
}
            /* End main procedure                  */
  :
            /* Window procedures/dialog procedures */
  :
            /* Start of help hook procedure        */

BOOL EXPENTRY HelpHook( HAB hab, SHORT sMode, SHORT idTopic,
                        SHORT idSubTopic, PRECTL prcPosition );
{
  :
            /* Use the information in the HelpHook input */
            /* parameters to give context-sensitive help */
  :
}
            /* End of help hook procedure          */
```

A help hook procedure is a piece of application-provided code that is called whenever the user requests help. The user can do this by pressing the F1 key or by selecting a help menu item or push-button. The WM_HELP message is still sent to the active window, but is seen by a modal loop if one is active.

For example, the frame window procedure sees WM_HELP because the frame is usually the active window. If the client window has the focus, the frame passes WM_HELP to the client window. The client window procedure may need to process the WM_HELP message in order to help supply the context information for the help hook. Otherwise, the frame window procedure calls the help hook procedure, passing the information required for the context-sensitive help in the help hook parameters.

For example, if the parent of the window with the focus is the client window, sMode is HFM_WINDOW, idTopic is the frame window identifier, and idSubTopic is the focus window identifier.

For more information, see the description of HelpHook in the *Control Program Programming Reference*.

It is the responsibility of the help hook to provide the help information, including the general help, contextual help, help index, and help on keys. The help information should be displayed in a window. The help information should be defined as a resource, so that it can be translated into other languages.

Hooks Function Calls

WinSetHook (WMSHOO)
Installs a hook.

WinReleaseHook (WMRHK)
Removes a hook.

WinCallMsgFilter (WMCMFI)
Calls a message filter hook.

The Atom Manager

This section describes the functions used for creating and destroying atom tables; for adding, finding, and deleting atoms within an atom table; and for accessing the string name and usage count associated with an atom.

The atom manager provides a mechanism for converting a string (atom name) into a 16-bit word (atom) that may be used as a constant to represent the string in various application and system data structures. By converting strings to atoms once, you can save space when the same string must be kept in various data structures. It saves time when comparing for a particular string, as you only need to convert the search string to an atom once, then run over your data structures doing word compares with the atoms stored in the data structures.

The Atom Table

The atom manager uses an atom table to hold the strings associated with atoms along with the control structures needed to probe the table to see if a string is already there. Atom tables have the following properties:

- The maximum length of an atom name is 255 characters. A zero length string is not a valid atom name.
- When searching for an atom name in an atom table, case is significant and the entire string must match (no substring matching is performed). The search is performed using a hash table. Collisions are resolved using chaining.
- The maximum amount of data that can be stored in an atom table is 64KB. This includes any control data needed by the atom manager to manage the atom table (see below).
- The maximum number of string atoms allowed is 16 384. The values of string atoms can range from 0xC000 through 0xFFFF.
- Associated with each string atom is a usage count that is incremented each time the atom is "added" to the table and decremented each time the atom is "deleted" from the table. This allows multiple users of the same atom string to coexist without destroying each other's atoms.
- The atom manager control data consists of 6 bytes of overhead for each string atom (not including the string itself); plus 2 bytes of overhead for each bucket in the hash table; plus 16 bytes of overhead for the atom table itself.

Integer Atoms

In addition to string atoms, there is a special kind of atom called an integer atom that has its own set of properties:

- Integer atoms can range from 0x0001 to 0xBFFF. The range of integer atoms is separate from the range of string atoms, so the two types of atoms can be used in an application without fear of overlap.
- The string representation of an integer atom is #ddddd where the ddddd are decimal digits. Leading zeros are ignored. These strings must always be specified in the system code page (CP 850).

- There is no usage count or storage overhead associated with an integer atom.

Integer atoms are useful for predefined system constants exported by a dynamic link package, as they behave exactly like atoms, except that they have no overhead. A good example of where integer atoms are used is in the Presentation Manager-registered window classes. Application-defined window class names are strings that the system converts into atoms for the purpose of detecting if the same class name is being defined more than once. Therefore, the Presentation Manager-registered window classes can also be expressed as atoms. Making them integer atoms allows the atoms to be expressed as compile-time constants in the Presentation Manager header file; the application can reference these classes and create windows with them without having to have a string constant in their data segment; there is no run time overhead in the system for remembering these atoms.

Using Atoms

Both types of atoms can be specified by a single far pointer that can be interpreted in one of four ways:

1. lpString -> "string" ; string-atom name
2. lpString -> "#ddddd" ; integer atom specified as a string
3. lpString -> "!",atom ; integer or string atom passed indirectly
4. lpString = FFFF atom ; integer or string atom passed directly

Note that these representations put a constraint on the first character. If it matches the number sign (#) or exclamation mark (!) in the system code page (CP850), the remainder of the string is treated differently.

Atom Management Function Calls

WinCreateAtomTable (WMCATT)
Creates an empty atom table of the specified size.

WinDestroyAtomTable (WMDATT)
Destroys an atom table.

WinAddAtom (WUADAT)
Adds an atom to an atom table.

WinFindAtom (WMFNAT)
Finds an atom in an atom table.

WinDeleteAtom (WMDELA)
Deletes an atom from an atom table.

WinQueryAtomName (WMQANM)
Returns an atom name associated with an atom.

WinQueryAtomLength (WMQALN)
Returns the length of the atom name associated with an atom.

WinQueryAtomUsage (WMQAUS)
Returns the number of times an atom has been used.

The Heap Manager

A heap is a memory segment that contains other memory objects. The memory objects are allocated and freed using the heap management functions. (The functions are not supported in COBOL/2 and FORTRAN/2.) This section describes the functions for:

- creating, destroying, and reorganizing heaps
- allocating, reallocating, and freeing memory objects within a heap
- accessing the base address of a heap.

A heap handle is a 32-bit quantity that uniquely identifies a heap, the segment that contains the heap, and the offset within the segment of the beginning of the heap. The interpretation of the heap handle is implementation-dependent.

A heap has the following properties:

- It is a segment allocated by OS/2, either by default because it is the application's or dynamic link package's automatic data segment or it was explicitly allocated with DosAllocSeg.
- It may not be larger than 64KB.
- All pointers to memory objects within a heap are 16-bit offsets from the start of that segment. All memory objects are aligned on a ULONG boundary. This means that the contents of the low-order 2 bits of a returned pointer are available for use by the caller. WinAllocMem sets the bits to zero. WinReallocMem and WinFreeMem ignore the bits, but preserve them in the returned value.
- The heap manager does not remember the size of a memory object allocated within a heap. The caller must remember the size and specify

it whenever reallocating or freeing a heap memory object. The heap manager always rounds up a size parameter to the next multiple of 4 (ULONG alignment); therefore, the caller does not need to be concerned with the size of any extra space allocated to meet the alignment constraints.

- The contents of memory allocated with the heap manager is undefined (that is, it does not explicitly set it to all zeros).
- When allocating or reallocating a memory object within a heap, DosReallocSeg may be called to expand the segment to satisfy the request. The amount of growth is bounded below by the size of the request and bounded above by the minimum growth amount, which is a parameter to the WinCreateHeap function.
- The heap manager maintains an array of free lists indexed by size. This lets the allocator find a free block quickly. Because the array occupies memory, there is a method for setting the size of the array when the heap is created. The minimum and default size of the array is one entry, called the non-dedicated free list. This entry is the head of a linked list of free blocks, in some undefined order. Additional entries in the free list array are called dedicated free lists.

 The WinCreateHeap function lets the caller specify the minimum and maximum object sizes that are to have their own dedicated free lists. Allocating and freeing objects within these minimum and maximum sizes that have their own free list is very quick, as no search is required if there is already a free block of the requested size. A linear search of the non-dedicated free list is performed only if a block of the requested size cannot be found on a dedicated free list.
- A function is provided that returns the size of the largest free block within a heap. It has the option of trying to to make a free block that is larger than a specified amount by reorganizing the heap.
- There is no overhead for allocated or free memory objects except for that imposed by the ULONG alignment (maximum of 4 bytes, average of 2 bytes). The only other overhead imposed by the heap manager is the size of the heap control block, which contains the free list array and a few other control words.
- If a zero size object is allocated, it occupies a minimum of 4 bytes so that it has a unique address to return to the caller.
- The ability of a heap to share depends upon the segment that contains the heap. Heaps contained within an application's data segment are private to that application. Heaps contained within a dynamic link package's data segment are either private to each process (instance data segment) or shared across all processes (global data segment). Segments explicitly allocated with DosAllocSeg are share or private, depending upon the setting of the shareInd parameter to DosAllocSeg. Segments allocated by WinCreateHeap are allocated shareable. Note that shared segments are not allowed to shrink, which means that heaps within a shared segment are also not able to shrink.
- The heap manager does not try to provide mutual exclusion between multiple threads of execution that try to call the heap manager with the same heap handle. The caller must ensure that this does not occur.

Moveable Heaps

A special type of heap, called a moveable heap, allows the memory objects within the heap to move to reclaim fragmented heap space. All heaps are moveable in the sense that the segment that contains a heap can move, because of the selector-to-physical-address mapping provided by the 286 protected mode. Moveable heaps have the following properties:

- The moveable heap attribute is specified at the time the heap is created and lasts until the heap is destroyed.
- Allocated objects in a moveable heap have two extra words reserved at the beginning of each object: a handle word and a size word. The offset value returned by WinAllocMem and WinReallocMem is the offset of the first reserved word, the handle word.
- The size word is initialized to the exact size specified on the WinAllocMem call and does not include the 4 bytes allocated for the reserved words or any bytes allocated to meet the ULONG alignment constraints.
- The handle word is initialized to zero when an object is allocated. If it is set to a nonzero value by the caller, the value must be a 16-bit offset within the segment containing the heap.

This offset points to a word, called the handle value word. The low order bit of the handle word must always be zero, as it is used to identify free blocks. This allows the compaction algorithm in WinAvailMem to do a linear scan of the objects in the heap.

- The WinReallocMem and WinAvailMem functions are free to move blocks that have a nonzero handle word. Allocated objects whose handle word is zero are considered fixed and not moveable.
- When an object with a non-zero handle word moves, the handle value word is updated by the delta amount of the move. By adding a delta instead of storing the new address of the object, it minimizes the constraints upon the type of address in the handle value word. For example, the caller may store the address of the first byte after the two reserved words in the handle value word, in order to hide the reserved words from code that accesses the memory object.
- The caller has to reserve space for the handle value words. If the space is within a heap memory object, it should be a fixed-memory object with a handle word of zero. Otherwise, incorrect results will occur.
- The size parameter to the WinReallocMem and WinFreeMem functions is ignored for objects in a moveable heap, as the value of the size word is used instead.
- Objects within a moveable heap can move whenever the WinAvailMem function is called. Since this function is also called by WinAllocMem and WinReallocMem, objects can also move when these functions are called as well.

Why the Heap Manager Exists

The heap manager should always be used for Presentation Manager applications rather than the Memory Sub-Allocation Package (MSP), for the following reasons:

1. The MSP requires that the heap is always at the beginning of a segment. This means that it cannot be used to allocate memory out of the space reserved at the end of an application's automatic data segment by the HEAPSIZE keyword in the application .DEF file. The same is true for the automatic data segment associated with a dynamic link package.
2. The MSP does not maintain dedicated free lists, and thus is required to do a linear search of its non-dedicated free list whenever a memory object is allocated. Because the free list is kept in address-sorted order, the cost of the search can vary. The cost of a free can also vary. This in an implementation restriction, but adding dedicated free lists requires two API changes to support the concept well:
 a. The ability to control the number of dedicated free lists
 b. An API call to coalesce free blocks, because it cannot be done when a memory object is freed, as MSP does.
3. The MSP does not provide any mechanism for supporting moveable objects within a heap.

Heap Management Function Calls

WinCreateHeap
Creates a heap that can used for memory management.

WinDestroyHeap
Destroys a heap created by the WinCreateHeap call.

WinAvailMem
Returns the size of the largest free block on the heap.

WinAllocMem
Allocates memory from a heap.

WinReallocMem
Reallocates the size of a memory block on the heap.

WinFreeMem
Frees memory allocated by WinAllocMem.

WinLockHeap
Converts a heap handle to a far pointer.

Chapter 14. Customizing the User Interface

Presentation Manager provides several functions that allow the user to tailor some aspects of the system. These functions are provided by Desktop Manager and the Control Panel. This chapter describes how programs can access these functions and, in some cases, change them for the user. However, take care not to impose changes on the user unless they are to the user's advantage. In particular, if an application changes parameters, such as the warning beep, for its own purposes, it should reset them to their previous values before it closes.

The Initialization Files

The initialization files contain information about printers, queues, and system preferences set when Presentation Manager was installed, or set by the user from the Control Panel. The file is in binary format and is used to set up the user's profile when Presentation Manager is initialized.

Initialization for Different Environments

In a normal single-user system, a single initialization file is sufficient. But in a multiple-user or medialess system, more than one initialization file may be required. There are two logical classes of information normally contained in the initialization files:

- System-specific information, which applies to the hardware
- User information, which applies to the current user of the system.

To provide consistency across all environments, Presentation Manager separates these two classes of information into two separate initialization files. For ease of reading, in this chapter the user initialization file is referred to as OS2.INI, and the system initialization file as OS2SYS.INI. The actual paths and file names are normally specified in the PROTSHELL command in the CONFIG.SYS file, or are provided when an application wants to change the environment.

OS2SYS.INI contains the system and hardware information:

- Names of ports
- Names of printers and printer drivers
- Names of queues
- The spooler path.

OS2.INI contains the information that is specific to an individual user or terminal:

- Screen colors
- Default printer and queue
- Desktop Manager list
- Application settings.

Calls are provided to allow applications to access and change the two initialization files. These calls are described in "Function Calls for Initialization Files" on page 14-2.

Accessing Initialization Information

All data in the initialization files is stored in a two-level hierarchy of an application name and a keyname. System data is keyed from applications that have names starting with PM_.

The system application names that other applications can use are listed below. Details of the keynames and data associated with each application are given in the *Programming Reference: Volume 2.*

OS2.INI Application Names

PM_ControlPanel
Controls beep and logo display time.

PM_National
Controls country code pages and date, time, and currency display.

PM_Fonts
Describes font files that are available for public use.

PM_SPOOLER
'Printer' keyname defines the default printer.

OS2SYS.INI Application Names

PM_SPOOLER
Defines default printer queue.

PM_SPOOLER_PRINTER
Describes each installed printer.

PM_SPOOLER_QUEUE_DESCR
Available queue names.

PM_SPOOLER_QUEUE_DD
Queue names with associated default device drivers.

PM_SPOOLER_QUEUE_DDDATA
Available queue names.

PM_SPOOLER_PRINTER_DESCR
Recognized printer names.

Displaying a Logo Panel

When started by the user, many programs display a logo panel. This typically contains the name of the company or corporation that developed the product, the name of the product, and copyright information. To save the user's time, the Control Panel lets the user decide whether or not to have the logo panel displayed. The user can also specify for how long applications should display their logo panels. The Control Panel stores this information in the OS2.INI file under the application name PM_ControlPanel and the keyname LogoDisplayTime. Before your application displays a logo panel, it should query the OS2.INI file to find out if the user has specified a time limit.

When the user has not specified a time limit, there is no LogoDisplayTime keyname entry in OS2.INI, and the call returns a default value to indicate this. Alternatively, the user can specify that the logo panel should be displayed indefinitely. A value of −1 specifies that the logo should be displayed indefinitely, until a key is pressed. In both cases, your application should display its logo until the user responds.

A zero time period specifies that the logo should not be displayed. When the user has specified a time limit of zero, your application should not display its logo.

When the user has specified a time limit that is greater than zero, your application should display its logo for the duration of the time limit. It can do this by issuing a WinStartTimer (WISTAT) call, using the time limit as an input parameter. Your application must process the resulting WM_TIMER message. Use the WinStopTimer (WISTOT) call to stop the timer, then remove the logo panel from the screen, and display the application's main window.

Using Public Fonts

Query the PM_Fonts application without a keyname to get the names of any font files that are available for public use. The returned string contains the full path and names of any font files known to the system.

Alternatively, you can make any fonts that you install available to other users by registering them in the initialization files. If you register them in OS2SYS.INI, they are available to all users who use the same file for their OS2SYS.INI. If you register them in OS2.INI, they are available to all users who use the same file for their OS2.INI. Write to the initialization files using PM_Fonts as the application name and the font name as the keyname, together with the full path and name of the font file.

Function Calls for Initialization Files

The following calls can be used to query or write to either OS2SYS.INI, OS2.INI, or both. The file that is accessed is determined by the **hini** parameter of the call. A value of `HINI_USERPROFILE` accesses OS2.INI. A value of `HINI_SYSTEMPROFILE` accesses OS2SYS.INI. If `HINI_SYSTEMPROFILE` is specified when there is no OS2SYS.INI known to the system, the call looks in the OS2.INI file. `HINI_PROFILE` reads both initialization files, but writes to OS2.INI only.

If the **hini** value is not one of those described above, it must be the handle of a file that has been defined as an initialization file using the PrfOpenProfile (PROPPR) call.

One of the profile files must contain the system default program list, spooler defaults, and default appname/keyname pairs. One way to do this is to create the basis for at least one of the profile files using **MAKEINI.EXE**. MAKEINI takes three parameters:

IniFile
The full path and file name of the file to be created

Additions
The full path and file name of the resource file used to add default information

Deletes
The full path and file name of the resource file containing appname/keyname pairs to be deleted before the additions are made. This parameter is optional.

The following example creates a new profile file, MYNEW.INI, containing U.S. default information:

```
A:MAKEINI C:OS2\MYNEW.INI A:INIUS.RC
```

The calls that let you query or write to an initialization file are:

PrfQueryProfileString (PRQPST)
Returns a text string from an initialization file.

If an application name and keyname are supplied, the returned string is the information stored for that application/keyname pair.

If the keyname is NULL, the string contains all the keynames associated with the named application.

If the application name is NULL, the string contains all the application names in the initialization file.

PrfWriteProfileString (PRWPST)
Writes a text string to an application/keyname entry.

If the string is NULL, the entry for the given keyname is erased.

If the keyname is NULL, all the keynames for the given application are removed.

If the supplied initialization file, application name, or keyname do not exist, they are created by this call.

PrfQueryProfileSize (PRQPSZ)
Returns the size, in bytes, of an entry for a given application/keyname pair.

If the application name is NULL, the value is the size of the application name list.

If the keyname is NULL, the value is the size of the keyname list.

PrfQueryProfileData (PRQPDA)
Returns a string of binary data from an initialization-file entry.

If an application name and keyname are supplied, the returned string is the information stored for that application/keyname pair.

If the keyname is NULL, the string contains all the keynames associated with the named application.

If the application name is NULL, the string contains all the application names in the initialization file.

PrfWriteProfileData (PRWPDA)
Writes a string of binary data to an initialization file.

If the string is NULL, the entry for the given keyname is erased.

If the keyname is NULL, all the keynames for the given application are removed.

PrfOpenProfile (PROPPR)
Defines a file that can be used as OS2SYS.INI or OS2.INI. The file name is supplied as a parameter to the call.

PrfCloseProfile (PRCPR)
Removes the definition of the file name supplied with the call so that it can no longer be used as an OS2SYS.INI or OS2.INI file.

PrfReset (PRRES)
Resets and reinitializes Presentation Manager, using file names supplied with this call as the OS2SYS.INI and OS2.INI files.

PrfQueryProfile (PRQPR)
Gets the names of the current OS2SYS.INI and OS2.INI files.

The Installed Program List

For an OS/2 program to be recognized by Presentation Manager, it must have an entry in the installed program list. When a user adds an application to the Desktop Manager, this list is automatically updated. The user is asked to provide:

- A program title. This is displayed in the Desktop Manager window.
- The path and file name of the program.
- The name of the program's working directory, if it needs one.

- Any parameters that are passed to the program when it starts.

A series of calls are supplied with Presentation Manager that allow applications to automatically add themselves to the installed program list when they are installed. Other calls are provided that query, add programs to, or delete programs from, the installed programs list. These calls allow programs to check whether they are already included in the list, or to remove references to old versions before installing a new one. The calls are described below.

Installed-Program Function Calls

PrfAddProgram (PRADPR)
Adds the definition of a program to the installed list.

PrfChangeProgram (PRCPRG)
Replaces the information that is stored for a program.

PrfRemoveProgram (PRRPRG)
Removes a program definition from the installed program list.

PrfCreateGroup (PRCGRP)
Creates a new group in the installed list.

PrfDestroyGroup (PRDGRP)
Removes the definition of a given group from the installed programs list. The root group and protected groups cannot be destroyed.

PrfQueryDefinition (PRQDEF)
Gets information about a program or group in the installed program list. If a group name is supplied, a list of program titles is returned. If a program name is supplied, the program details are returned.

PrfQueryProgramCategory (PRQPC)
Returns the category of the given executable file name. The category can be Presentation Manager, full screen, or windowable.

PrfQueryProgramTitles (PRQPTI)
Gets type and title information for a given executable file or for a named group.

PrfQueryProgramHandle (PRQPHD)
Gets a list of the program handles associated with a given executable file name. This tells a caller if, and where, a given executable file is referenced in the program list.

The entries for the installed program list are kept in the OS2.INI file. They can therefore be accessed using the installation file function calls described in "Function Calls for Initialization Files" on page 14-2.

Starting and Terminating Installed Applications

Once a program has been installed, it can be started and terminated by another application. Two calls are provided to allow your application to do this:

WinInstStartApp (WMISAP)
Starts an installed application from a specified profile file. If you want your application to be able to terminate the new application, use the SAF_STARTCHILDAPP option. This defines the new application as a child of your application.

WinTerminateApp (WMSTOP)
Terminates an application that has been started by WinInstStartApp using the SAF_STARTCHILDAPP option. When the application has stopped, your application receives a WM_APPTERMINATENOTIFY message.

List of Running Programs

The user shell displays a list of running programs in the Task List window. The list is sometimes referred to as the "switch list", because the user can switch between the programs on the list. When a user starts a program, the user shell adds the file name of the program to the switch list.

Presentation Manager provides a series of calls that allow programs to query, add, change, or delete items in the switch list. A good convention for titles in the switch list is to use the title that is displayed at the top of your program's main window. (When your program has more than one main window, and you want the user to be able to switch to each window independently, you can add a switch list entry for each window.)

If your program adds itself to the switch list when it is started, it should remove itself from the list when it stops.

The calls that access the switch list are described below.

Switch List Function Calls

WinAddSwitchEntry (WMASWE)
Adds a switch list entry.

WinChangeSwitchEntry (WMCSWE)
Changes a switch list entry.

WinRemoveSwitchEntry (WMRSWE)
Removes a switch list entry.

WinSwitchtoProgram (WMSTP)
Makes the specified program the active program. This call is only acted on if it is issued by the application that is the current foreground process.

WinQuerySwitchEntry (WMQSEN)
Gets information about an entry in the switch list that is associated with the handle provided with the call.

WinQuerySwitchHandle (WMQSHD)
Gets the switch list handle belonging to a window.

WinQueryTaskSizePos (WMQTSP)
Gets the recommended size, position and status for the first window of a newly-started application. This information may come from an initialization file, but if no data is available, the system generates the values.

WinQuerySessionTitle (WMQSTI)
Gets the title under which a specified application is started, or added to the switch list.

WinQuerySwitchList (WMQSLI)
Gets information such as window handles and session identifiers for all the entries in the switch list.

Saving the Current State of Your Application

A user may want to save your application in its current state at the end of a session, and return to it at the start of the next session. For example, a text-editing application could save the name of the file that it is editing, together with the current line number.

Presentation Manager sends a WM_SAVEAPPLICATION message when it is about to close, so on receipt of this message, your application should save any variables it needs to restore itself to its current state when it is next started. You can save the variables in an initialization file or any other suitable file. When the application is started again, it should retrieve these variables and give the user the option to use them.

Note: COBOL and FORTAN applications do not receive *sent* messages, so the WM_SAVEAPPLICATION message has no effect.

Each application can have only one set of saved variables at any time. When a WM_SAVEAPPLICATION message is received, any previously saved variables must be replaced by the new data. When your application has saved its variables, it should call WinDefWindowProc (WMDWP) to pass the WM_SAVEAPPLICATION message to the default window procedure.

When Presentation Manager sends a WM_SAVEAPPLICATION message, it is passed from one application to another. Therefore it is important that your application does not delay processing the message. If you save controls such as message boxes and dialog boxes, or if you query the names of files during the save procedure, you could delay the processing of the message.

Querying System Values

There is a set of system constants defined by OS/2. Each constant has the prefix SV_, meaning system value. For example, the width and height of the display screen are held in SV_CXSCREEN and SV_CYSCREEN. You can query the system values using the WinQuerySysValue (WMQSYS) call. Some of the system values can be changed, using the WinSetSysValue (WMSSYS) call. All the SV_ values are listed in the *Control Program Programming Reference*.

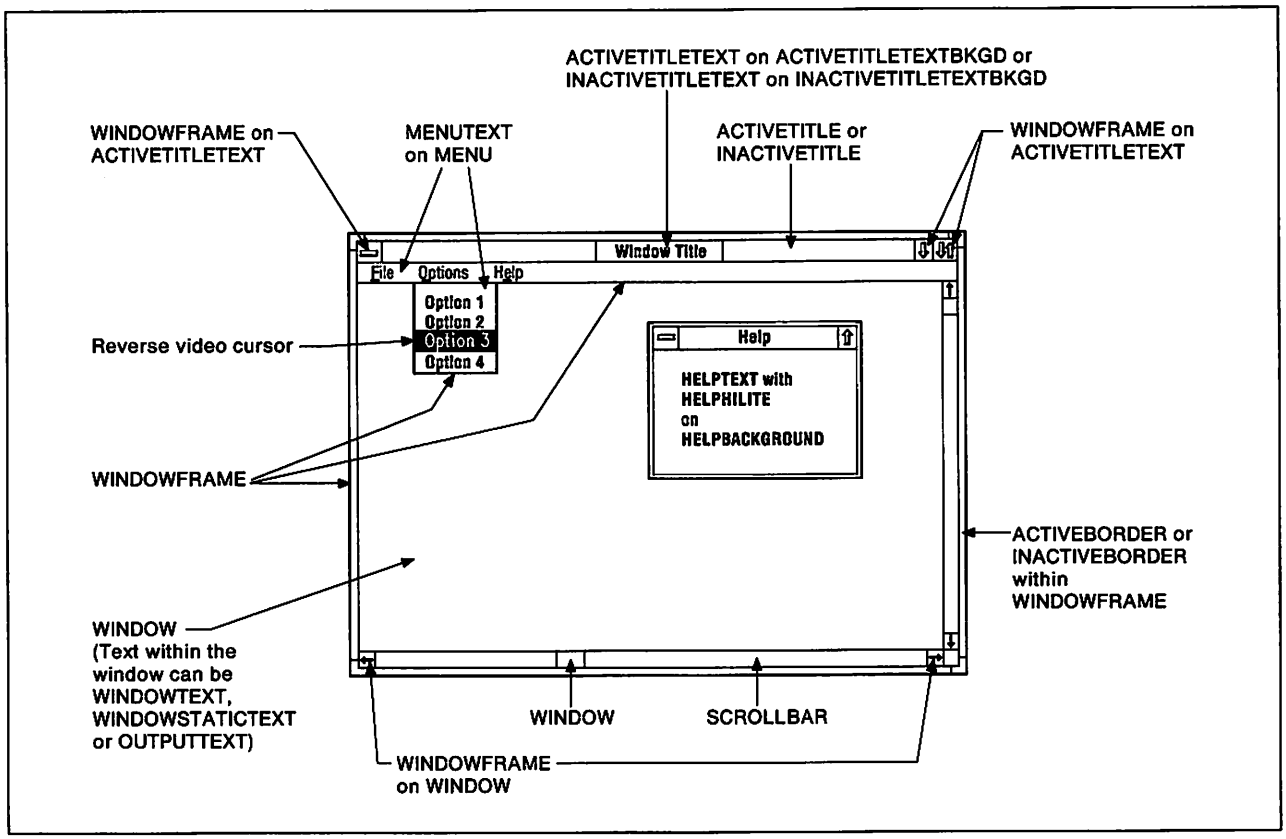


Figure 14-1. Use of System Color Indexes in a Window. Prefix the indexes with SYSCLR_ when you use them in your application.

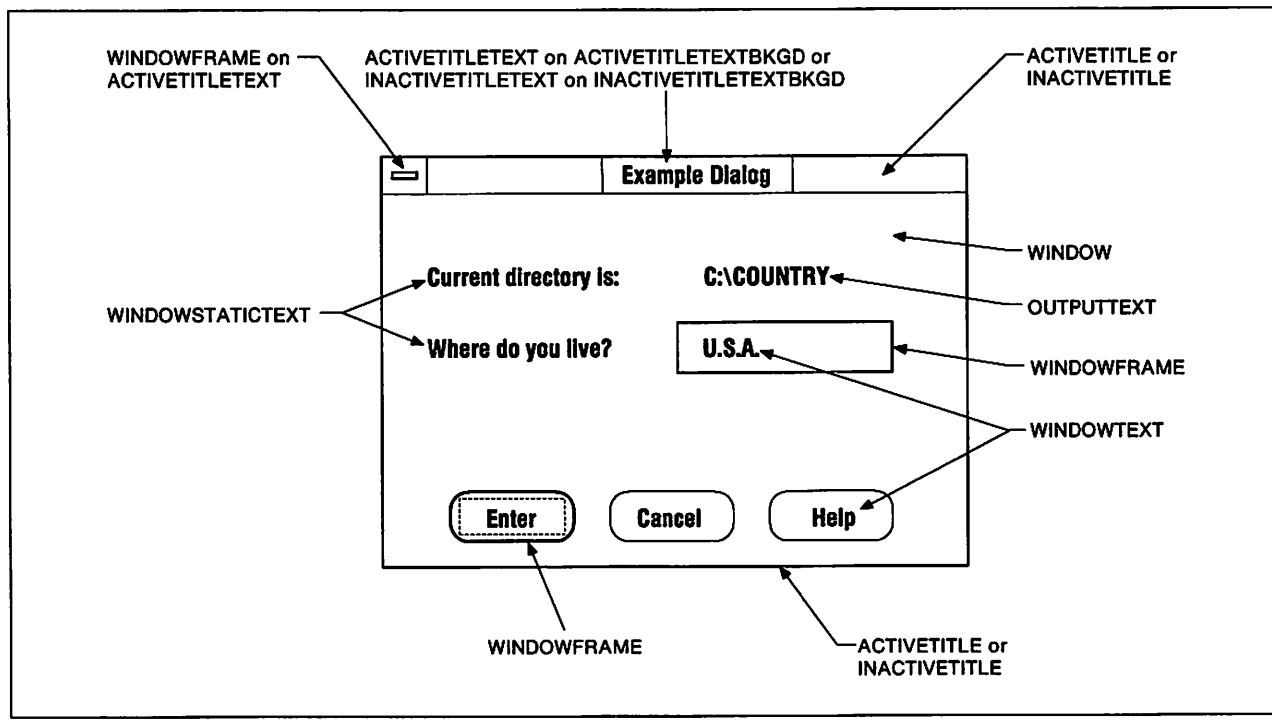


Figure 14-2. Use of System Color Indexes in a Dialog Box. Prefix the indexes with SYSCLR_ when you use them in your application.

Using the System Colors

When Presentation Manager is installed, the colors of its visual elements conform to the Common User Access. These colors are called the system colors. The OS/2-defined window classes also use the system colors. Figure 14-1 shows how the system color indexes are used in a window, and Figure 14-2 shows how they are used in a dialog box.

If you want your own application-defined visual components (objects you draw in the client window) to use the system colors, use the WinQuerySysColor (WIQSCO) call to query the color for each system color index. If the user does not change the system colors (they can be changed using the Control Panel), your application's user interface automatically conforms to the Common User Access.

Programmer's Responsibility

Although the system colors are applied automatically to most of the visual components, it is the programmer's responsibility to assign them to the following:

SYSCLR_APPWORKSPACE
: Background of those main windows that contain only child windows and no controls (for example, the File Manager main window).

SYSCLR_WINDOWSTATICTEXT
: Static text and instructions in windows.

SYSCLR_WINDOW
: Background of application windows.

SYSCLR_WINDOWTEXT
: Selectable or changeable text in main windows.

SYSCLR_OUTPUTTEXT
: Variable output information.

SYSCLR_WINDOWSTATICTEXT, SYSCLR_WINDOW, and SYSCLR_WINDOWTEXT are assigned automatically in dialog boxes.

If you use the Help Manager to create your help panels, they conform to the Common User Access. If you do not use the Help Manager, but you still want your application to conform to the Common User Access, use the default colors of the following indexes in your help panels:

SYSCLR_HELPBACKGROUND
: Help panel background color.

SYSCLR_HELPTEXT
: Help panel text.

SYSCLR_HELPHILITE
: Help panel highlighted text.

Part 3. Presentation Manager Applications – Using OS/2 Information Presentation Facility

Chapter 15. An Introduction to the OS/2 Information Presentation Facility

This chapter provides an overview of the features and characteristics of the OS/2 Information Presentation Facility (IPF) help interface and the work effort involved in implementing this interface for your Presentation Manager application.

Although Common User Access (CUA) guidelines are referred to in this chapter, the information provided here is not intended as a definitive discussion of all CUA recommendations for a help facility.

The User Requirement for Online Help

While running an application, the user sometimes requires help. For example, a user may need assistance in making a choice or in recalling the name of an application command. The application does not include this kind of information as part of an application window because it is not always needed. Nor is it very practical. In most cases, the information would not fit in the window.

Common User Access (CUA) guidelines recommend that an application make online help available at the user's request, and that the user help interface be presented in a consistent format.

The OS/2 Information Presentation Facility (IPF) is a programming tool that follows CUA guidelines for implementation of a help interface. Using IPF, you can develop an interface that provides not only general help for application windows, but also contextual help for fields within windows.

Access to Help from the Application Interface

CUA guidelines recommend that users be able to access general help for an application window by selecting **Help** as the last choice on the application action bar or by pressing the F1 key. Users should also be able to request field-level help by pressing the F1 key or by clicking on a help pushbutton within a dialog box or message box window.

When a user selects the **Help** choice from the application action bar, a pull-down should be displayed, whose appearance is similar to the help pull-down shown in Figure 15-1.

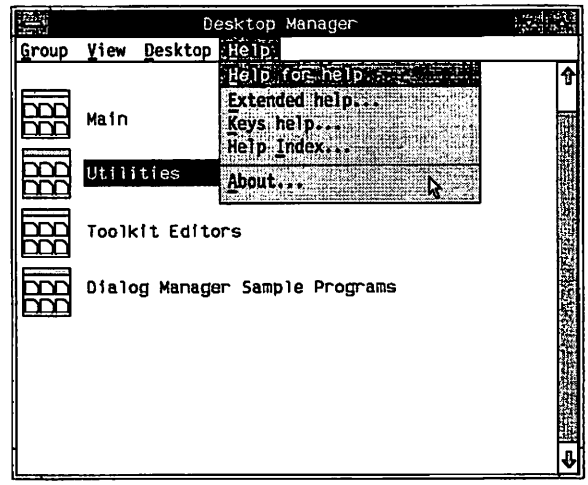


Figure 15-1. Application Action Bar Help Pull-Down

Developing a Help Interface

Implementing the IPF help interface for an application requires two different development efforts:

- Developing the code that creates an instance of the help interface and communicates with IPF and the Presentation Manager.
- Developing a library of help information that is referenced by IPF in response to a user request.

In some organizations, a technical writer (sometimes referred to as the information developer) develops the help panels at the same time as the programmer (sometimes referred to as the application developer) develops the code that associates the help panels with the user's requests for help.

An overview of each of these development efforts is contained in the following sections. To better understand the concepts described, you can examine the files required to build the HELP1 Presentation Manager sample application.

Coding the Application

When a user requests help from the help interface, the Presentation Manager responds by sending a message to the application window's procedure.

To call the IPF help hook in response to the request, your application does the following:

- Initializes the IPF initialization structure
- Creates a help instance
- Associates the help instance with the application window chain.

Associating the help instance with the window chain identifies the window with which IPF communicates, so the appropriate help panel can be displayed.

Table structures in application memory or in resource files identify panel resources in the IPF help library. The help table associates each application window with its corresponding help subtable and the panel ID of its extended help panel. The help subtable associates each entry field within an application window with the panel ID of its help panel.

For descriptions of function calls, IPF messages, data structures, and resource definitions for the help interface, see Chapter 16, "Developing Application Code for the Help Interface" on page 16-1.

Creating a Library of Help Panels

A library of help panels is created from ASCII text files containing the responses to user help requests. The text of help panels includes "tags" the writer imbeds as the help information is developed. These tags comprise the IPF Tag Language, which is the mechanism that defines various characteristics of text presentation for a panel displayed in the help text window. For more information on the IPF Tag Language, see "IPF Tag Language Markup" on page 17-1.

When the information is ready for viewing, the IPF Compiler (IPFC) is used to convert the tagged help files into a format suitable for display. IPFC can distinguish the tags from the help text in which they are imbedded, because each tag starts with a colon.

After reviewing the compiled help panels, the writer can edit the source files and recompile them.

Each help panel is given a panel ID as part of the tag that creates the heading displayed in the title bar of the help text window. These panel IDs are also used in application code.

Descriptions of how these tags are used to create help panels are contained in Chapter 17, "Developing Help Information" on page 17-1. For an alphabetic listing of tags and their function descriptions, see Appendix H, "Information Presentation Facility Tags" on page I-1.

A User's Perspective of the Help Interface

This section acquaints you with the functions and characteristics of the IPF help user interface. The information contained in this section can be used for reference by both the application developer and the help panel information developer.

Help Window Characteristics

When a user requests help from an application window, IPF displays the main help window. The characteristics of the main help window are:

- System menu icon
- Title of the application's help interface
- Help action bar.

The main help window cannot be minimized.

Within the main help window is shown the help text window. The help text window contains the response to the user's request for help. The characteristics of the help text window are:

- System menu icon
- Title of the selected help panel
- Vertical scroll bar.

The title displayed in the title bar is the one created by the author of the help panel. The vertical scroll bar is active only if the information in the help panel does not fit in the help text window.

When the main help window is first opened, it is positioned so that it covers the smallest portion of the application window as possible. The help text window is opened as maximized within the main help window.

After the main help window is opened, it can be moved and resized by the user, as can the help text window. If the user resizes the help text window to make it larger or smaller, the text within the window is reformatted to fit the new window size.

A subsequent user help request for which help is available, causes the information in the help text window to be replaced. The help text window cannot be closed. To view a previously viewed response to a help request, the user selects **Previous** from the **Options** help action bar pull-down or presses the Esc key.

List box windows are displayed when the user selects any of the following help action bar pull-down choices:

- Viewed pages, under **Options**
- Contents, under **Options**
- Help index, under **Help**.

The results of a search are also displayed in a list box window. List box windows are about one-third the size of help text windows and so do not completely overlay a previously displayed help text window.

List box windows differ from the help text window in that they can be closed, either by selecting **Close** from the system menu icon or by pressing Ctl+F4. If a list box window is made smaller, the list is not reformatted.

Selecting Minimize from the system menu displays a unique icon for a help text window, Contents window, Viewed pages window, Help index window, or Search results window.

Help Action Bar

The help action bar, which is part of the main help window, is shown in Figure 15-2. This action bar is suitable for the help requirements of most Presentation Manager applications. Because it conforms to CUA guidelines, it is recommended that you use it. However, if your application has help requirements not met by the help action bar, IPF allows you to customize the action bar template. See "A Customized Action Bar" on page 16-12 for programming information.

Selecting **Services**, **Options**, or **Help** from the action bar reveals a pull-down with a list of entries that can be selected. Each entry has an associated accelerator key or key combination. Accelerator keys are shown as right-aligned in the pull-down.

Services Action Bar Pull-Down

Selecting the Services action bar choice displays a pull-down containing a list of services available to users of the help interface. This list is shown in Figure 15-2.

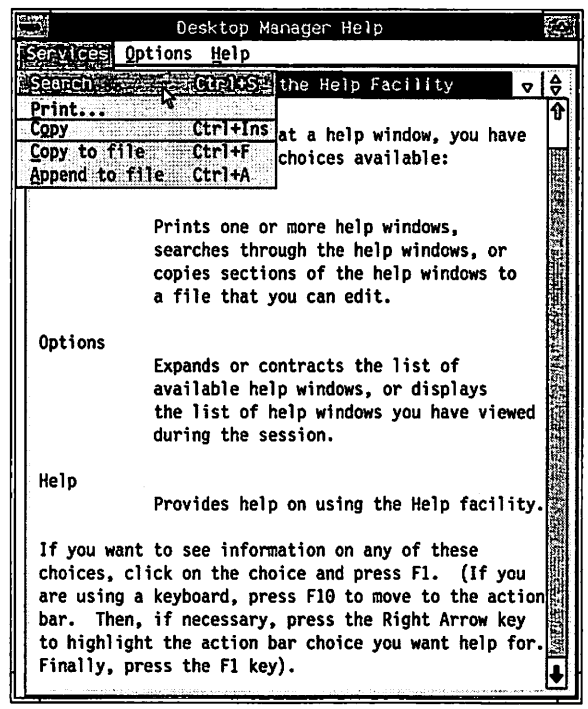


Figure 15-2. Services Pull-Down

Search: This selection displays the dialog box shown in Figure 15-3 on page 15-4. The user can select any of the choices to have a literal search performed for an ASCII text string containing letters, numbers, blank spaces, and special characters.

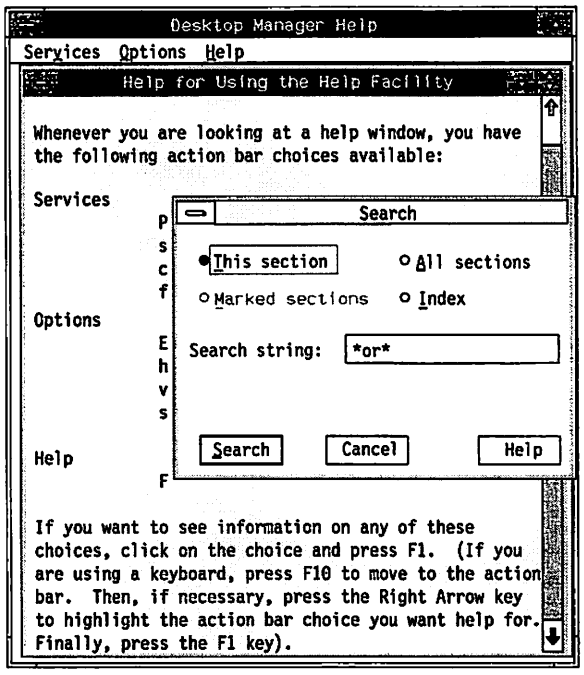


Figure 15-3. Search Dialog Box Window

Global characters may be used preceding and/or following the search string, for example:

```
*val*
```

Following are descriptions of Search dialog box selections:

This section: Searches the currently displayed help text window and highlights all occurrences found (if any) of the search string.

Marked sections: Searches the help panels whose titles were marked in the contents window. Before selecting Marked sections, the user has to select contents and mark the help titles to be searched. If no help titles are marked, the marked sections choice is grayed.

Sections are marked with the mouse by pressing the Ctl key and pressing button 1 of the mouse. Sections are marked with the keyboard by using the cursor keys to highlight the item and then pressing the spacebar. The same key sequences are used to unmark the selection.

If the search is successful, IPF displays a list of the panels where occurrences of the search string were found (if any). The search string is shown in the search results window title.

If the user marks sections, they remain highlighted. The user can unmark them or mark additional sections to continue the search.

All sections: Searches the entire help library and displays a list of the panels where occurrences of the search string were found (if any). The search string is shown in the window title bar.

Index: Searches the help index and displays a list of index entries in which occurrences of the search string were found (if any).

Note: If synonym search help is provided, a synonym search follows the literal search. For more information, see "Synonyms — The :isyn. Tag" on page 17-4.

If no occurrences of a search string are found, the **String not found** message is displayed.

Print: This selection displays the dialog box shown in Figure 15-4. The user can select any of the choices to print help information.

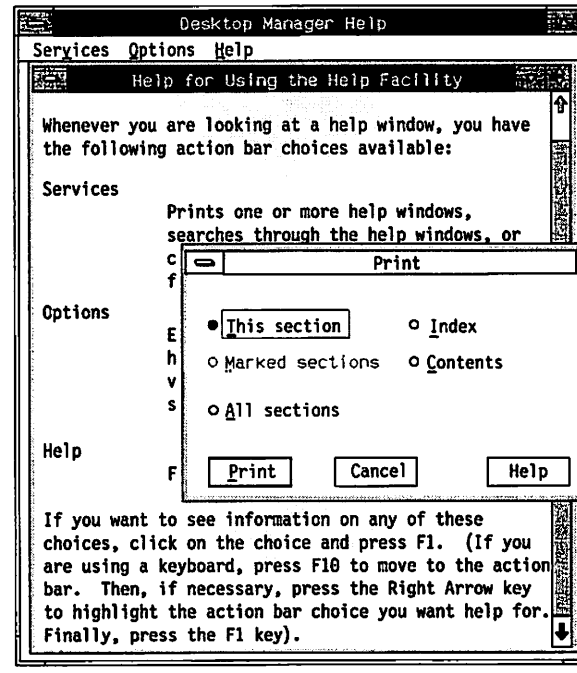


Figure 15-4. Print Dialog Box Window

Following are descriptions of Print dialog box selections:

This section: Sends the contents of the help text window being viewed to be printed.

Marked sections: Sends the contents of the help panels whose titles were marked in the contents window to be printed. Before selecting Marked sections, the user has to select contents and mark the help titles to be printed. If no help titles are marked, the Marked sections choice is grayed.

Sections are marked with the mouse by pressing the Ctl key and pressing button 1 of the mouse. Sections are marked with the keyboard by using the cursor keys to highlight the item and then pressing the spacebar. The same key sequences are used to unmark the selection.

All sections: Sends the entire help library to be printed.

Index: Sends the help index list to be printed.

Contents: Sends the help contents list to be printed.

The help text the user selects to be printed, is submitted to the Print Spooler, so the user can continue to work.

Copy Choices in the Services Pull-Down

When one of the copy choices is selected, the help text is copied, either to the system clipboard or a file. If the help information being copied includes character graphics, they are copied but may not format correctly when pasted to an editor. Bit-map graphics cannot be copied.

Copy: This selection copies the help currently being viewed to the system clipboard. The user can then select the **Paste** option from the action bar of the OS/2 System Editor (or any other editor with this capability) to view or edit the help information.

Copy to file: This selection creates the file TEXT.TMP and copies the help being viewed to the file. This file is placed in the user's current directory. If TEXT.TMP already exists in the user's directory, it is replaced.

Append to file: This selection creates the file TEXT.TMP and copies the help currently being viewed to the file. This file is placed in the user's current directory. If TEXT.TMP already exists in the user's directory, it is appended.

Options Action Bar Pull-Down

Selecting the Options action bar choice displays a pull-down containing the choices shown in Figure 15-5.

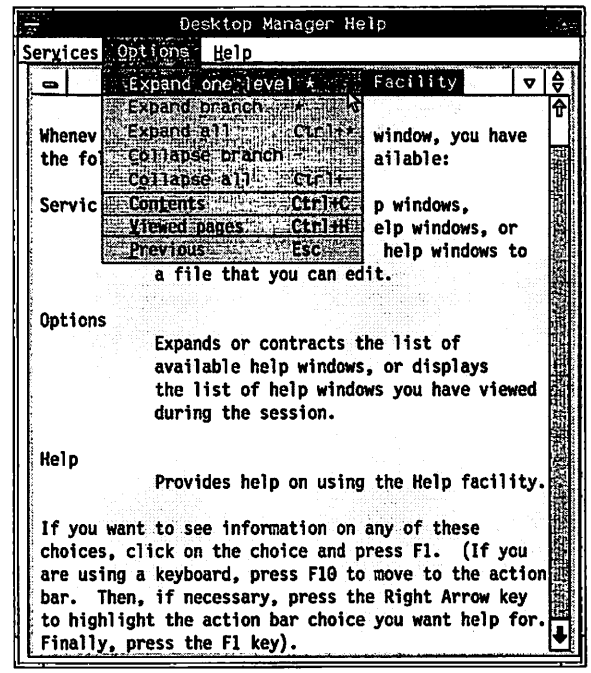


Figure 15-5. Options Pull-Down

The first five choices are active when the **Contents** choice has been selected. If the contents window is not active, these choices are grayed. These choices are used to expand and contract a tree-structured table of contents. A tree-structured help contents is created if more than one heading level is specified with IPF heading tags when the help panels are created. (For a description of heading tags, see "Defining a Panel with IPF Markup" on page 17-2.)

If the contents is expandable, + icons appear to the left of contents entries. When an entry with a + icon is expanded one level, the next level of entries immediately below the selected entry is displayed, and the + icon is replaced by a – icon. The user also can click on the icons to expand or contract the contents.

Expand one level: This selection expands the first subordinate level beneath the selected entry.

Expand branch: This selection expands all subordinate levels beneath the selected entry up until the next entry whose level is equal to the selected entry's level.

Expand all: This selection displays the entire tree structure of the contents.

Collapse branch: This selection contracts all subordinate levels beneath the selected entry up until the next entry whose level is equal to the selected entry's level.

Collapse all: This selection displays only the highest level entries in the contents.

Contents: This selection displays a list of application help panel titles in the order of their sequence in the help library file, merged with the list of system help panel titles. The system list follows the application list.

Viewed pages: This selection displays a list of all panels viewed during the current help session. Help titles are listed in the order that panels were viewed. If a panel is viewed more than once, its title appears as many times in the sequence as it was viewed.

The maximum number of entries that can be displayed in the 'Viewed pages' list box window is 50.

Previous: This selection displays the previously viewed help panel. Each time **Previous** is selected, the previously viewed panel is displayed until the first panel viewed in the current help session is shown. After the first panel viewed is displayed, clicking on **Previous** ends the current help session.

Help Action Bar Pull-Down

Selecting the **Help** action bar choice displays a pull-down containing the choices shown in Figure 15-6.

The Help pull-down is described in the following sections.

Help for help: This selection displays general information describing the help interface and how it is accessed by the user. This help panel is shipped with IPF and can be replaced by information that is specific to your application.

See "Help for Help Request" on page 16-5 for programming information relating to the replacement of Help for help.

Extended help: This selection displays general information describing the contents of the application window from which the user has requested help.

It is possible the user will request help (by pressing F1 or selecting the **Help** pushbutton) when the cursor is positioned on an item in the application window for which there is no specific help. In a case like this, the application can respond by displaying the panel for Extended help.

See "When No Help Is Available" on page 16-5 for programming information relating to Extended help.

Keys help: This selection displays information describing application key assignments. Providing Keys help is an application responsibility. A simple help panel can be created that lists each key combination that is assigned to an application function, followed by a brief description of the function performed.

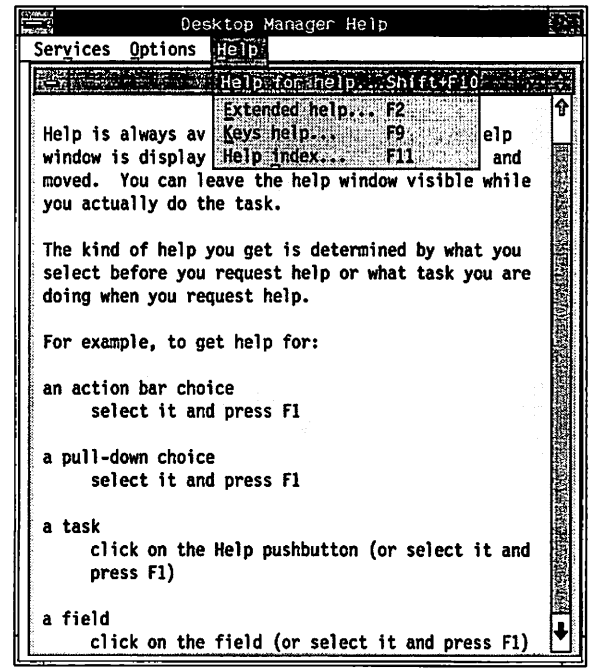


Figure 15-6. Help Pull-Down

Help index: This selection displays an alphabetic list of help topics for which index entries have been included in help panel information. System help topics are merged with application help topics. Index entries for system help topics are included as

secondary entries under one primary entry, "help window entries."

Tutorial: This selection is included in the help pull-down if your application tells IPF it has created a Presentation Manager tutorial application. When the user selects Tutorial, the application receives a message from IPF that this choice has been made. This allows the application to start the tutorial. For programming information related to this choice, see "Tutorial Name" on page 16-1.

Chapter 16. Developing Application Code for the Help Interface

This chapter is intended for the application developer. It describes the interactive help interface that can be developed with the OS/2 Information Presentation Facility (IPF).

Creating a Help Instance

Each time the user requests help from an application window, either by pressing the F1 key, selecting a Help pushbutton, or selecting a choice from the help pull-down, the Presentation Manager responds by sending a message to the application window's window procedure.

To have IPF install a help hook to trap user help requests, your application has to tell the Presentation Manager it wants IPF to create a help instance to handle user help requests. The Presentation Manager code that calls the IPF help hook is in the Presentation Manager default window procedure (either WinDefWindowProc for a standard window, or WinDefDlgProc for a dialog box window).

The calls relating to the creation of the help instance are as follows:

WinCreateHelpInstance
Creates a help instance for an application. After its creation, IPF handles all requests for help from the application.

WinAssociateHelpInstance
Associates the help instance with a chain of application windows. This enables IPF to identify the particular help instance that is providing help function to your application.

WinDestroyHelpInstance
Ends the window chain's association with the help instance.

The **WinCreateHelpInstance** call passes the HELPINIT structure defined in the PMHELP.H include file to the Presentation Manager. The PMHELP.H file also contains the error codes returned in the event of an unsuccessful call. You include this file in your source code by using the INCL_WINHELP define statement.

WinCreateHelpInstance returns a handle to the help instance, which you must store in a HWND variable, for use with the rest of the IPF function calls. (See "IPF Function Calls" on page 16-14 for a summary of these calls.)

IPF responds to the WinCreateHelpInstance call by installing its help hook and initializing for help processing.

In the Main Procedure of the HELP1 Sample Program, shown on page 16-4, the control data parameter *hmiHelpData* in WinCreateHelpInstance is a pointer to the initialization structure. The anchor block handle is the one returned to the application from the Presentation Manager WinInitialize call.

Initializing the HELPINIT Structure

Before you call WinCreateHelpInstance, you must allocate memory for and initialize the HELPINIT structure. This structure defines values that IPF needs to create the help instance. Some of the values can be changed by your application after initialization.

The HELPINIT structure and the help table structures referred to by IPF during help processing are contained in the PMHELP.H file. Following are descriptions of HELPINIT structure fields. (For a description of help table structures, see "Help Tables Used by IPF" on page 16-10.)

Structure Length: The length of the initialization structure identifies the version of OS/2 IPF being used.

Tutorial Name: If the initialization structure contains a pointer to a tutorial name, IPF provides a 'Tutorial' choice in the help pull-down. If this pointer is NULL, this means either the application help interface does not include a tutorial, or, if it does, the tutorial is referenced from a help panel.

If the user selects the 'Tutorial'choice in the help pull-down, IPF sends the HM_TUTORIAL message to the application, so it can start the tutorial.

DLL Resource Fields: The pointer to the help table can be an address in memory, or it can be a resource ID indexing into a dynamic link module. If the help table (and its corresponding subtables) are in a dynamic link module, the module must be loaded by the application with a call to DosLoadModule prior to initialization. The same holds true for the accelerator table and action bar template resources; that is if the application has customized the help action bar and if its resources reside in a DLL.

If these fields are null, this means resources were appended to the application's executable file or, in the case of the action bar, that the default is to be used.

After initialization, a new help table can be accessed by IPF from a dynamic link module already loaded by calling WinLoadHelpTable or by sending the HM_LOAD_HELP_TABLE message.

A new help table also can be accessed from memory by calling WinCreateHelpTable or by sending the HM_CREATE_HELP_TABLE message.

Window Title: A pointer to the name of the title for the main help window is specified in the initialization structure. This name can be changed after initialization by sending the message HM_SET_HELP_WINDOW_TITLE.

Show Panel IDs: Setting the CMIC_SHOW_PANEL_ID flag is useful during the development stage of the help interface. This causes panel IDs to be appended to the beginning of the help panel title in the title bar of the help text window.

If the flag is set to CMIC_HIDE_PANEL_ID, panel IDs will not be shown. (These constants are defined in the PMHELP.H file.)

After initialization, the displaying of panel IDs can be toggled with the HM_SET_SHOW_PANEL_ID message.

Names of Help Libraries: Help library names are the names of the .HLP Files created by the IPF Compiler (IPFC), as described in "Compiling Source Files" on page 17-17. When IPF needs to search for a help panel, it looks for these library names in the path set by the HELP environment variable. If IPF cannot find a library name in this path, it then searches the current directory.

After initialization, specify names for help libraries with the HM_SET_HELP_LIBRARY_NAME message. If multiple libraries are specified, library names must be separated by a blank space.

See page 16-3 for the HELP1.C entries for the IPF initialization structure.

Associating the Instance with the Window Chain

After an application creates a help instance, it must associate the instance with the application window chain by calling to **WinAssociateHelpInstance**. IPF uses the active window handle passed by this call to index into the help table to find the help panel that should be displayed for the application window.

An IPF instance can be associated with any application window that has a frame. Once the association of an IPF instance with the application window chain is made, help can be requested for any application window in the chain.

The format of the WinAssociateHelpInstance call is shown in the HELP1 Main Procedure on page 16-4.

Allowing IPF Access to the Active Window

By sending the HM_SET_ACTIVE_WINDOW message, the window with which IPF communicates is specified explicitly. If this message is not sent, IPF asks for the active window using the Presentation Manager WinQueryActiveWindow call. IPF then searches the parent/owner chain for the current association. This window is used to index into the help table. The window IPF communicates with, is the one specified as the *hwndApp* parameter of the WinAssociateHelp Instance call.

Ending the Help Instance

To end the current help instance, the application calls **WinDestroyHelpInstance**, passing the handle of the help instance to be destroyed. The format of WinDestroyHelpInstance is shown on page 16-5 in the HELP1 Main Procedure.

The parameter *hwndHelpInstance* is the handle to the IPF instance returned from the WinCreateHelpInstance call.

Example Code: HELP1 Main Procedure

```
HELPINIT hmiHelpData;                   /* Help initialization structure*/
HWND     hwndHelpInstance;              /* Handle to Help window        */
/*********************  Start of main procedure  **********************/
void cdecl main(  )
{
  HMQ  hmq;                             /* Message queue handle         */
  HWND hwndClient;                      /* Client area window handle    */
  HWND hwndFrame;                       /* Frame window handle          */
  QMSG qmsg;                            /* Message from message queue   */
  ULONG flCreate;                       /* Window creation control flags*/

  hab = WinInitialize( NULL );          /* Initialize PM                */
  hmq = WinCreateMsgQueue( hab, 0 );    /* Create a message queue       */

  WinRegisterClass(                     /* Register window class        */
     hab,                               /* Anchor block handle          */
     "MyWindow",                        /* Window class name            */
     MyWindowProc,                      /* Address of window procedure  */
     CS_SIZEREDRAW,                     /* Class Style                  */
     0
     );
  /*********************************************/
  /* IPF Initialization Structure              */
  /*********************************************/
                                        /* size of initialization structure*/
  hmiHelpData.cb                      = sizeof(HELPINIT);
                                        /* store HM return code from init. */
  hmiHelpData.ulReturnCode            = NULL;
                                        /* no tutorial program             */
  hmiHelpData.pszTutorialName         = NULL;
                                        /* indicates help table is defined */
                                        /* in the RC file.                 */
  hmiHelpData.phtHelpTable            = (PVOID)(0xffff0000 | MAIN_HELPTABLE);
                                        /* action bar is not tailored      */
  hmiHelpData.hmodAccelActionBarModule = NULL;
  hmiHelpData.idAccelTable            = NULL;
  hmiHelpData.idActionBar             = NULL;
                                        /* help window title               */
  hmiHelpData.pszHelpWindowTitle      = "Help1 Help Window";
                                        /* help table in not in a DLL      */
  hmiHelpData.hmodHelpTableModule     = NULL;
                                        /* help panels ID is not displayed */
  hmiHelpData.usShowPanelId           = NULL;
                                        /* library with help panels        */
  hmiHelpData.pszHelpLibraryName      = "Help1.HLP";
  /*********************************************/
  /* Create Instance of IPF                    */
  /*                                           */
  /* pass Anchor Block handle and address of   */
  /* IPF initialization structure              */
  /*********************************************/
```

```
hwndHelpInstance = WinCreateHelpInstance(hab, &hmiHelpData);
if (!hwndHelpInstance)
   {
   WinMessageBox( HWND_DESKTOP, HWND_DESKTOP,
                  (PSZ) "Help Not Available",
                  (PSZ) "Help Creation Error",
                  1,
                  MB_OK | MB_APPLMODAL | MB_MOVEABLE);
   }
else
   {
   if (hmiHelpData.ulReturnCode)
      {
      WinMessageBox( HWND_DESKTOP, HWND_DESKTOP,
                     (PSZ) "Help Terminated Due to Error",
                     (PSZ) "Help Creation Error",
                     1,
                     MB_OK | MB_APPLMODAL | MB_MOVEABLE);
      WinDestroyHelpInstance(hwndHelpInstance);
      }
   }


flCreate = FCF_STANDARD &            /* Set Frame Control Flags to   */
           ~FCF_SHELLPOSITION;       /* standard except for shell    */
                                     /* positioning.                 */

hwndFrame = WinCreateStdWindow(
            HWND_DESKTOP,           /* Desktop window is parent     */
            0L,                     /* No  frame styles             */
            &flCreate,              /* Frame control flag           */
            "MyWindow",             /* Client window class name     */
            "",                     /* No window text               */
            0L,                     /* No special class style       */
            NULL,                   /* Resource is in .EXE file     */
            ID_WINDOW,              /* Frame window identifier      */
            &hwndClient             /* Client window handle         */
            );

WinSetWindowPos( hwndFrame,         /* Shows and activates frame    */
                 HWND_TOP,          /* window at position 50, 100,  */
                 50, 100, 200, 200, /* and size 200, 200.           */
                 SWP_SIZE | SWP_MOVE | SWP_ACTIVATE | SWP_SHOW
               );

/*******************************************/
/* Associate Instance of IPF               */
/*                                         */
/* pass handle to Help Window and handle   */
/* to frame window with which Help is      */
/* to be associated.                       */
/*******************************************/
if (hwndHelpInstance)
  {
  WinAssociateHelpInstance (hwndHelpInstance, hwndFrame);
  }
```

```
/*********************************************************************/
/* Get and dispatch messages from the application message queue      */
/* until WinGetMsg returns FALSE, indicating a WM_QUIT message.     */
/*********************************************************************/
  while( WinGetMsg( hab, &qmsg, NULL, 0, 0 ) )
    WinDispatchMsg( hab, &qmsg );

    /***************************************/
    /* Destroy Instance of IPF             */
    /*                                     */
    /* on termination the active IPF       */
    /* instance should be destroyed.       */
    /***************************************/
  if (hwndHelpInstance)
    {
    WinDestroyHelpInstance (hwndHelpInstance); /* Destroy Help Instance      */
    }

  WinDestroyWindow( hwndFrame );            /* Tidy up...                  */
  WinDestroyMsgQueue( hmq );                /* and                         */
  WinTerminate( hab );                      /* terminate the application   */
}
/*********************  End of main procedure  *********************/
```

Processing Help Requests

IPF communicates with the active window. This communication is accomplished through messaging. The application may need to do some of its own processing in response to these messages.

Help for Help Request

When the user selects 'Help for help' from the help pull-down, a WM_COMMAND is sent to the application's window procedure.

If the application has created its own 'Help for help' panel, it responds by sending the HM_REPLACE_HELP_FOR_HELP message with the panel's ID. If the application chooses to use the 'Help for help' panel provided by IPF, it responds by sending the HM_DISPLAY_PANEL with NULL in both parameters.

In the Window Procedure of HELP1 shown on page 16-8, HELP1 has chosen to display the IPF 'Help for help' panel.

Keys Help Request

When the user selects 'Keys help' from the help pull-down, an HM_KEYS_HELP message is sent by the application to IPF. In response, IPF sends an HM_QUERY_KEYS_HELP message to the application. The application returns the panel ID of the keys help panel.

In the Window Procedure of HELP1 shown on page 16-8, HELP1 provides the ID of its keys help panel.

When No Help Is Available

A user may request help by pressing F1 when the cursor is positioned on an item for which no field-level help is available. In such a case, IPF sends the HM_HELPSUBITEM_NOT_FOUND message to the application. The application can return FALSE, to have IPF display the panel for extended help. If the application returns TRUE, there is no response to the user request.

If the application does not respond to HM_HELPSUBITEM_NOT_FOUND, the default window procedure causes the help panel for 'Extended help' to be displayed, if one is available.

IPF Messages

Following is a summary of the messages sent by IPF and the application, in response to user help requests. For detailed descriptions of the parameters and returns for these messages, see the "Information Presentation Facility Messages" chapter in the *Presentation Manager Programming Reference: Volume 2.*

Messages Sent by IPF

These messages are sent by IPF to the application when the following events occur:

HM_ACTION_BAR_COMMAND
The user has selected a customized action bar item. This message allows the application to perform an application function related to the item.

HM_ERROR
An error has resulted from a user interaction. This message allows the application to display its own error message and maintain a consistent user interface for all application messages. For more information on error handling, see "IPF Error Handling" on page 16-14.

HM_EXT_HELP_UNDEFINED
An extended help panel has not been defined for the application window.

HM_HELPSUBITEM_NOT_FOUND
The user has requested help on a field for which no related entry can be found in the help subtable. The application can ignore this message, display its own panel, or send HM_DISPLAY_HELP to IPF.

HM_INFORM
The user has selected a hypertext field defined in the help panel with the **inform.** attribute of the **:link.** tag. This allows the application to perform an application-specific function. For example, the application may choose to start a tutorial.

HM_QUERY_KEYS_HELP
The user has selected the Keys help function from the Help pull-down. The application responds by returning the panel ID of the Keys help panel.

HM_TUTORIAL
The user has selected the Tutorial function from the Help pull-down. The application can respond by calling DosExecPgm to start its own tutorial.

Messages Sent by the Application

These messages are sent by the application to IPF and have the following meanings:

HM_CREATE_HELP_TABLE
Use the new help table indicated by this address in memory.

HM_DISMISS_WINDOW
Remove the help window associated with the last active window.

HM_DISPLAY_HELP
Display the help panel indicated.

HM_EXT_HELP
Display the extended help panel for the active application window.

HM_HELP_CONTENTS
Display the help Contents window.

HM_HELP_INDEX
Display the help index window.

HM_KEYS_HELP
Display the keys help panel in the help window. IPF responds by sending HM_QUERY_KEYS_HELP to the application, and the application returns the panel ID of the keys help panel.

HM_LOAD_HELP_TABLE
Use the new help table indicated by the help table resource ID.

HM_REPLACE_HELP_FOR_HELP
Display the panel indicated by panel ID as a replacement for Help for help default.

HM_SET_ACTIVE_WINDOW
Change the active application window with which the IPF help window is associated.

HM_SET_HELP_LIBRARY_NAME
Replace the list of help libraries specified in the initialization structure with a new list and search these libraries in response to subsequent help requests.

HM_SET_HELP_WINDOW_TITLE
Change the title of the main help window.

HM_SET_SHOW_PANEL_ID
Display or hide panel IDs of help panels. Displaying panel IDs is useful during the development stage of the help interface.

Example Code: HELP1 Window Procedure

```
/********************* Start of window procedure **********************/
MRESULT EXPENTRY MyWindowProc( HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
  USHORT command;                       /* WM_COMMAND command value     */
  HPS    hps;                           /* Presentation Space handle    */
  RECTL  rc;                            /* Rectangle coordinates        */
  POINTL pt;                            /* String screen coordinates    */

  switch( msg )
  {
    case WM_CREATE:
      /*****************************************************************/
      /* Window initialization is performed here in WM_CREATE processing*/
      /* WinLoadString loads strings from resource file                */
      /*****************************************************************/
      WinLoadString( hab, NULL, IDS_1,     STRINGLENGTH, sz1 );
      WinLoadString( hab, NULL, IDS_2,     STRINGLENGTH, sz2 );
      WinLoadString( hab, NULL, IDS_3,     STRINGLENGTH, sz3 );
      WinLoadString( hab, NULL, IDS_4,     STRINGLENGTH, sz4 );
      strcpy( szString, sz1 );      /* Copy text into szString*/
      break;

    case WM_COMMAND:
      /*****************************************************************/
      /* For the Commands pull down, when the user chooses option 1, 2, */
      /* or 3, the text string is set to 1, 2, or 3, and               */
      /* WinInvalidateRegion sends a WM_PAINT message.                 */
      /* For the Exit pull down, the application posts itself a WM_CLOSE*/
      /* message if Exit is chosen, or resumes if Resume is chosen.    */
      /*****************************************************************/
      command = SHORT1FROMMP(mp1);      /* Extract the command value    */
      switch (command)
      {
          /*************************************************************/
          /* Help Display Utility                                      */
          /*                                                           */
          /* It is possible to modify this code to create a simple     */
          /* help panel display utility.  You can modify the options   */
          /* processing with the following code to display the help    */
          /* contents and help index.  It would also, be appropriate   */
          /* to add code to prompt for the help library name and use   */
          /* a HM_SET_HELP_LIBRARY_NAME message to pass this data      */
          /* to IPF.  By simply copying the help library you want to   */
          /* display to Help1.hlp you will have a help display utility */
          /*                                                           */
          /* case ID_OPTION1:                                          */
          /*   WinSendMsg(hwndHelpInstance,HM_HELP_CONTENTS, 0L, 0L);  */
          /*   break;                                                  */
          /* case ID_OPTION2:                                          */
          /*   WinSendMsg(hwndHelpInstance,HM_HELP_INDEX, 0L, 0L);     */
          /*   break;                                                  */
          /*                                                           */
          /*************************************************************/
```

```
      case ID_OPTION1:
        strcpy( szString, sz2 );
        WinInvalidateRegion( hwnd, NULL, FALSE );
        break;
      case ID_OPTION2:
        strcpy( szString, sz3 );
        WinInvalidateRegion( hwnd, NULL, FALSE );
        break;
      case ID_OPTION3:
        strcpy( szString, sz4 );
        WinInvalidateRegion( hwnd, NULL, FALSE );
        break;
      case ID_EXITPROG:
        WinPostMsg( hwnd, WM_CLOSE, 0L, 0L );
        break;

      /*****************************************************/
      /* DISPLAY HELP_FOR_HELP                             */
      /*                                                   */
      /* In order to display the system help for help panel*/
      /* send a HM_DISPLAY_HELP message to the help        */
      /* manager with 0L in both parameters                */
      /*****************************************************/
      case ID_HELP_FOR_HELP:
        if (hwndHelpInstance)
          {
          WinSendMsg( hwndHelpInstance, HM_DISPLAY_HELP, 0L, 0L);
          }
        break;

      default:
        return WinDefWindowProc( hwnd, msg, mp1, mp2 );
    }
    break;

    /*************************************************/
    /*     QUERY_KEYS_HELP                           */
    /*                                               */
    /* If the user requests KEYS HELP from the help  */
    /* pull-down, IPF sends the HM_QUERY_            */
    /* KEYS_HELP message to the application, which   */
    /* should return the panel id of the keys_help   */
    /* panel, or a 0 to tell the HM to do nothing    */
    /*************************************************/

  case HM_QUERY_KEYS_HELP:
    return( (MRESULT) KEYS_HELP_PANEL_ID);
    break;
```

```
    /************************************************/
    /*     HM_ERROR                                 */
    /*                                              */
    /* If an error occurs using IPF, an HM_ERROR msg  */
    /* will be sent to the application.  At this point*/
    /* you can display a message box and destroy the  */
    /* help instance.  This code can be replaced to   */
    /* perform the error processing which you desire. */
    /************************************************/

  case HM_ERROR:
    WinMessageBox( HWND_DESKTOP, HWND_DESKTOP,
                   (PSZ) "Help Terminated Due to Error",
                   (PSZ) "Help Error",
                   1,
                   MB_OK | MB_APPLMODAL | MB_MOVEABLE);
    WinDestroyHelpInstance(hwndHelpInstance);
    break;

  case WM_ERASEBACKGROUND:
    /*********************************************************************/
    /* Return TRUE to request PM to paint the window background       */
    /* in SYSCLR_WINDOW.                                                 */
    /*********************************************************************/
    return (MRESULT)( TRUE );
  case WM_PAINT:
    /*********************************************************************/
    /* Window contents are drawn here in WM_PAINT processing.         */
    /*********************************************************************/
                                  /* Create a presentation space  */
    hps = WinBeginPaint( hwnd, NULL, &rc );
    pt.x = 50; pt.y = 50;          /* Set the text coordinates,    */
    GpiSetColor( hps, CLR_NEUTRAL );       /* colour of the text,   */
    GpiSetBackColor( hps, CLR_BACKGROUND ); /* its background and    */
    GpiSetBackMix( hps, BM_OVERPAINT );    /* how it mixes,         */
                                           /* and draw the string...*/
    GpiCharStringAt( hps, &pt, (LONG)strlen( szString ), szString );
    WinEndPaint( hps );                    /* Drawing is complete   */
    break;
  case WM_CLOSE:
    /*********************************************************************/
    /* This is the place to put your termination routines             */
    /*********************************************************************/
    WinPostMsg( hwnd, WM_QUIT, 0L, 0L );  /* Cause termination       */
    break;
  default:
    /*********************************************************************/
    /* Everything else comes here.  This call MUST exist              */
    /* in your window procedure.                                         */
    /*********************************************************************/

    return WinDefWindowProc( hwnd, msg, mp1, mp2 );
  }
  return FALSE;
}
/*********************  End of window procedure  **********************/
```

Help Tables Used by IPF

One of the entries in the help initialization structure passed by WinCreateHelpInstance is the address of the help table. The help table and its corresponding subtables are used by IPF to associate a Presentation Manager field ID and its corresponding contextual help panel with the extended help panel of the window in which the field is located.

IPF uses the help table to find the extended help panel ID and the help subtable associated with an application window. Each application window has a corresponding help subtable, which contains an entry for each field in the window that has a contextual help panel.

When the user requests help on a field in the application window, IFP uses the help subtable associated with the field to find the panel ID of the contextual help panel for the field. The help subtable can also store optional entries relating to application-specific information.

The maximum size of the help table is 64KB. The number of help subtables is limited to 16 000.

Help table and help subtable structures are contained in the PMHELP.H file. Descriptions of these structures can be found in the "Data Types" chapter of the *Presentation Manager Programming Reference: Volume 1.*

IPF supports two methods of defining help tables and help subtables. They can be allocated in memory, or they can be defined as resources.

Defining Help Tables in Memory

By defining help tables and subtables in memory, you can change dynamically a single entry in the help table. You can add a new panel ID to be associated with a field or add fields that are to be associated with existing panels.

After the help table structure is initialized, the application can pass IPF the address in memory of the new help table, either by sending the HM_CREATE_HELP_TABLE message from its window procedure, or by calling **WinCreateHelpTable**.

Help Table in Memory

The help table contains the structure defined in the PMHELP.H file for each application window. This structure holds the following information:

- Application window ID
- Address of the window's subtable
- Panel ID of the window's extended help panel.

These entries are integers. The last entry in the list contains a NULL for each entry type, to indicate the end of the list.

Figure 16-1 contains an example of a help table for an application.

```
HELPSUBTABLE        table1, table2, table3,
                    table4, table5;

HELPTABLE     helpTableEntry[] =
{
  APP_WIND_1,  &table1, idExtHelp1,
  APP_WIND_2,  &table2, idExtHelp2,
  APP_WIND_3,  &table3, idExtHelp3,
  APP_WIND_4,  &table4, idExtHelp4,
  APP_WIND_5,  &table5, idExtHelp5,
  0,           NULL,    NULL
};
```

Figure 16-1. Example of a Help Table in Memory

Help Subtable in Memory

The help tablesubtable contains the structure defined in the PMHELP.H file for each application window. This structure holds the following information for each field in the application window:

- Field ID from which user requests help
- Panel ID of the contextual help panel associated with the field
- Optional, application-specific integers.

The last entry in the structure contains the word length for each field entry. The minimum number of words is two, which is the default.

Figure 16-2 on page 16-11 contains an example of a help subtable for an application window that has six fields.

```
HELPSUBTABLE        HelpSubTable [] =
{
  2,
  FIELD_ID_1, IDRES_HELP1,
  FIELD_ID_2, IDRES_HELP2,
  FIELD_ID_3, IDRES_HELP3,
  FIELD_ID_4, IDRES_HELP4,
  FIELD_ID_5, IDRES_HELP5,
  FIELD_ID_6, IDRES_HELP6,
  0,          0
};
```

Figure 16-2. Example of a Help Subtable in Memory

Defining Help Tables as Resources

If help tables are defined as resources, they can be bound to the application's executable file or they can reside in a dynamic link library module.

If help tables are defined as resources in a dynamic link library, the application must call DosLoadModule to load the DLL before it calls WinCreateHelpInstance. When the application calls WinCreateHelpInstance, it passes the handle to the DLL and the resource ID of the help table in the HELPINIT structure.

The application can load a new help table residing in the DLL by either sending the HM_LOAD_HELP_TABLE message from its window procedure, or by calling **WinLoadHelpTable**. The application passes the handle to the DLL and the resource ID of the new help table.

HELPTABLE Resource

A HELPTABLE resource contains a HELPITEM entry for each application window, dialog box, and message box for which help is provided.

Each entry of a HELPTABLE resource contains:

- HELPITEM keyword
- Application window ID
- ID of the HELPSUBTABLE resource
- Panel ID of the extended help panel.

All referenced HELPSUBTABLE resources must reside in the same .RES file as the HELPTABLE resource.

HELPSUBTABLE Resource

A HELPSUBTABLE resource contains an entry for each item that can be selected in an application window. Each of these items is assumed to be a child window of the application window identified in the HELPTABLE resource. The HELPSUBTABLE should contain a single SUBITEMSIZE and a HELPSUBITEM for each control, child window, and menu item.

Each entry of a HELPSUBTABLE resource contains:

- HELPSUBITEM keyword
- Field ID
- Panel ID of the field's help panel
- Optional, application-defined integers.

The integer ID of the field can be a control, menu item, or message box ID. The ID specified must be unique within the table. An ID of X'FFFF' is reserved for use by IPF.

The SUBITEMSIZE keyword is used to identify the size in words of each HELPSUBITEM. (All entries must be the same length.) If this value is specified, it must be greater than or equal to 2. If this value is not specified, it defaults to 2.

The optional integers value allows the writer of the resource script file to append additional integers to the end of each HELPSUBITEM for application-specific use.

Figure 16-3 on page 16-12 contains an example of .RC source for defining a HELPTABLE and its related HELPSUBTABLE resources.

Defining Window Resources

The following sections describe window resources you can define for the help interface.

Help Pull-Down

Common User Access (CUA) guidelines recommend that all application windows with action bars include a help pull-down. A description of this help pull-down is found in "Help Action Bar Pull-Down" on page 15-6 The help application action bar choice and corresponding pull-down menu is defined in your resource file.

```
HELPTABLE TABLE_1
BEGIN
HELPITEM parentwindow1, SUBTABLE_1,
  extendedhelppanel1
HELPITEM parentwindow2, SUBTABLE_2,
  extendedhelppanel2
END

HELPSUBTABLE SUBTABLE_1
[SUBITEMSIZE subitemsize1]
BEGIN
  HELPSUBITEM FIELD_ID1, helppanel1 [,
    integer1, ...n]
  HELPSUBITEM FIELD_ID2, helppanel2 [,
    integer1, ...n]
END

HELPSUBTABLE SUBTABLE_2
[SUBITEMSIZE subitemsize2]
BEGIN
  HELPSUBITEM FIELD_ID3, helppanel3 [,
    integer1, ...n]
  HELPSUBITEM FIELD_ID4, helppanel4 [,
    integer1, ...n]
END
```

Figure 16-3. Example of Help Tables in a Resource Script File

Figure 16-4 on page 16-13 contains action bar pull-down definitions from the HELP1 resource script file.

Help Pushbutton

If your application has a dialog or message area you may want to include the Help pushbutton in the function key areas of these secondary application windows (dialog boxes). To define the help pushbutton, use the Presentation Manager button style BS_HELP and BS_NOPOINTERFOCUS. The BS_HELP style causes Presentation Manager to call IPF when the user selects this help pushbutton. The BS_NOPOINTERFOCUS style enables the Presentation Manager to determine the field for which the user requested help. For more information on pushbuttons, see "Immediate Action – Pushbuttons" on page 7-8.

Defining a Command Entry Field

To provide command help for an entry field of your application, you can define the field as a command entry field by including the ES_COMMAND style, as defined by the Presentation Manager. Panels containing command help must be tagged with the **:icmd** tag.

For more information on entry field styles, see *Programming Reference: Presentation Manager Related Information.*

A Customized Action Bar

A help action bar template is shipped with IPF. The template is in the HMTAILOR.RC file. Included in the template is the help pull-down. You can customize the action bar by adding pull-downs and choices to the help action bar template. See "Specifying Action Bar and Pull-Down Choices" on page 5-9 for a description of how to add these items to an action bar.

The command values of the actions added by the application must be between X'7F00' and X'7FFF' for its commands. When an action bar or pull-down choice you have added is selected by the application user, IPF sends the HM_ACTION_BAR_COMMAND message to the currently active application window. The low-order word of *param1* contains the command value of the selected item.

The accelerator table maps function keys to commands on help panels. This table also is contained in the HMTAILOR.RC file. If you add a choice to the action bar that maps to a key on the keyboard, you must also add an entry to the accelerator table for that choice. See "Accelerators" on page 5-13 for a description of the accelerator table entry format. IPF functions depend on the entries that already exist in the shipped accelerator table. They must not be altered. The command value specified in the accelerator table entry must be the same command value that was specified for the associated action in the action bar template.

```
MENU    ID_WINDOW PRELOAD
BEGIN
  SUBMENU "~Options", ID_OPTIONS
  BEGIN
    MENUITEM "Option ~1\tCtrl+a", ID_OPTION1, MIS_TEXT
    MENUITEM "Option ~2\tCtrl+b", ID_OPTION2, MIS_TEXT
    MENUITEM "Option ~3\tCtrl+c", ID_OPTION3, MIS_TEXT
    MENUITEM SEPARATOR
    MENUITEM "E~xit \tF3",       ID_EXITPROG, MIS_TEXT
  END
  SUBMENU     "~Help",              ID_HELP
  BEGIN
    MENUITEM   "~Help for help...", ID_HELP_FOR_HELP
    MENUITEM   "~Extended help...", SC_HELPEXTENDED,MIS_SYSCOMMAND
    MENUITEM   "~Keys help...",     SC_HELPKEYS    ,MIS_SYSCOMMAND
    MENUITEM   "Help ~index...",    SC_HELPINDEX   ,MIS_SYSCOMMAND
  END
END                             /* For Extended Help, Help Keys, and      */
                                /* Help Index, allow the system to process */
```

Figure 16-4. Pull-Down Definitions from the HELP1.RC Resource Script File

If the HMTAILOR.RC file is changed, you must compile it using the resource compiler and attach it to the executable file. If the executable file is a DLL, you must load it before calling WinCreateHelpInstance. by calling DosLoadModule. Identify the handle to IPF in the *hmodAccelActionBarModule* field in the initialization structure. When this field is NULL, IPF uses the default action bar.

The HMTAILOR.RC file includes the HMTAILOR.H file.

Creating More Than One Help Instance

It is possible to create multiple instances of IPF. To determine which instance of IPF help requests should be sent to, IPF traces the parent window chain, while it exists, and then the owner chain, until an instance associated with one of the windows is found. The parent chain is traced first until it is NULL or HWND_DESKTOP, and then the owner chain will be traced. If a parent of the owner window exists and no instance has been found the trace begins with the parent chain again.

The help instance associated with an application window can also be queried by calling **WinQueryHelpInstance**.

Installing an Application-Defined Help Hook

If the application's window procedure does not respond to the Presentation Manager WM_HELP message, the Presentation Manager default window procedure automatically handles the message with no application code.

If, however, the application's window procedure responds to the Presentation Manager's WM_HELP message, it must do the application-required processing and then call WinDefWindowProc or WinDefDlgProc. The value returned from the application processing the WM_HELP message should be the value returned from the WinDefWindowProc call or the WinDefDlgProc call.

When you use IPF for your application, we recommend that you *do not* install your own help hook. If you choose to do so, however, you must install the help hook prior to creating the IPF help instance. The IPF help hook procedure returns TRUE, allowing no further help hook chaining. Your application-installed help hook must return FALSE to enable IPF to process the help requests.

IPF Function Calls

Following is a summary of the calls associated with IPF functions. For detailed descriptions of the parameters for these calls, see the *Presentation Manager Programming Reference: Volume 1.*

WinCreateHelpInstance
Creates a help instance for an application. After its creation, IPF handles all requests for help from the application.

WinAssociateHelpInstance
Associates the help instance with a chain of application windows. This enables IPF to identify the particular help instance that is providing help function to your application.

WinDestroyHelpInstance
Ends the window chain's association with the help instance.

WinCreateHelpTable
Identifies or changes the pointer to the help table in application memory.

WinLoadHelpTable
Identifies or changes the handle of the module containing the help table resource and the resource ID of the help table.

WinQueryHelpInstance
Traces the window chain to find the IPF instance associated with the chain.

IPF Error Handling

Following are the two ways in which an application that has created a help instance can receive errors from IPF.

Return Value from a Call or Message

After the application initiates communication with IPF by sending messages, if an error occurs during message processing, the error is returned as the value of the message.

If an error occurs when an application communicates with IPF using a call, the application should use WinGetLastError to get the value of the error.

HM_ERROR Message

The application can also receive error notification from IPF by means of the HM_ERROR message. When an error is initiated by the user of the application, IPF sends this message to the active application window.

Errors can occur while IPF is responding to messages from a polling loop generated by a user action. Because the application is unable to be notified of the error by IPF if it is returned as the value of processing the message, IPF sends the HM_ERROR message to the active application window.

Chapter 17. Developing Help Information

This chapter is intended for the person who is developing the help information for the application's help interface.

The Information Developer's Perspective of the Help Interface

The writer of responses to help requests (the information developer) usually follows the same process in building and testing of the help interface as the application developer. However, the information developer may perceive the interface not as code, but as units of information.

Planning the Help Interface

Planning for the development of help information requires you to become acquainted with the characteristics of the help interface and identifying the parts of the application for which you will develop help. (For a description of the help interface, see "A User's Perspective of the Help Interface" on page 15-2.)

Common User Access (CUA) guidelines recommend that users be able to access general help for each application window and contextual help for each entry field or selection choice in a dialog box window. IPF provides this support. Figure 17-1 shows an example of contextual help provided by the *Desktop Manager* application of the OS/2 Presentation Manager. The help information in the help window is displayed when the user requests help from the the Copy Program dialog box window, either by pressing the F1 key or selecting the **Help** pushbutton.

In some organizations, planning of help information begins even before the first prototype of the application is made. The information developer reviews the specification for the application to assess its function and design — what the windows will look like, their relationship to one another, and how the user will interact with the windows.

After you have determined application requirements, you can begin writing responses to help requests, using IPF Tag Language markup to define the information as panels. A panel is the text unit associated by IPF with an application window or entry field. (For more information on panels, see "Defining a Panel with IPF Markup" on page 17-2.)

Later on in the development cycle, as panels are written and added to the interface, you may want to test the effectiveness of the help provided.

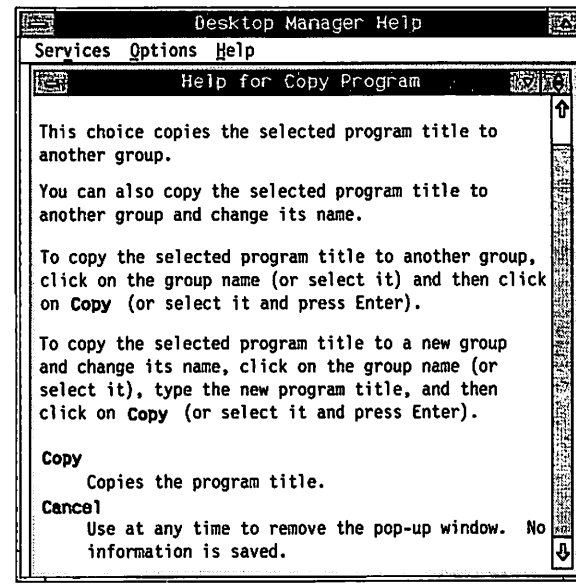


Figure 17-1. Desktop Manager Example of Help

IPF Tag Language Markup

The IPF Tag Language consists of instructions for display of online information. These instructions are sometimes referred to as "markup." This term is carried over from when typewritten pages were marked up with handwritten instructions to the typesetter, indicating the typographical composition of the printed page. Nowadays, most typesetting is done by computer, so the markup is in a language the computer understands.

People often associate the concept of typesetting only with printed information. However, typesetting instructions specified with IPF markup can produce online typography in a Presentation Manager text window that is equal in quality to typeset printed information. For descriptions of this markup, see "Text Presentation" on page 17-6.

In addition to typesetting operations, the IPF Tag Language also has markup that performs more sophisticated functions, such as linking to additional information. See "Linking from Hypertext Phrases — The :link. Tag" on page 17-13.

After help information has been written and marked up, it is ready to be compiled. The IPF Compiler (IPFC) reads the markup and translates it into an IPF library format, so that panels can be displayed in the help text window. The compiler can distinguish the markup imbedded within the text of help information, because each occurrence of markup starts with a colon (:) and ends with a period (.). Immediately following the colon is the tag name, which indicates the operation to be performed. The tag is a mnemonic, making it easy for you to associate its name with its function.

For more information about IPFC, see "Compiling Source Files" on page 17-17.

Other Types of Markup

In addition to tags, IPFC recognizes other types of markup. For example, it recognizes the following control words, which allow you to:

.im	Imbed a file.
.*	Put comments in your file.

As you can see, control words begin with a period (.). Examples of their use will be given later on.

IPFC also recognizes definitions of special characters (or symbols) that cannot be entered from the keyboard. Symbols begin with an ampersand (&).

For an alphabetic list of IPF tags and control words, see Appendix H, "Information Presentation Facility Tags" on page I-1. The list of symbols you can use is found under "Symbols."

Characteristics of Tag Markup

The markup for some tags is very simple. For example, the markup that indicates a new paragraph is:

```
:p.
```

This markup is translated as, "End the current line, insert a blank line, and start the text that follows at the beginning of the next line."

A tag may have one or more attributes associated with it. An attribute contains additional information relating to a tag's operation. The attribute has a name, which may have a keyword or value assigned to it. Some attributes are optional and may have defaults if they are not included in the markup; other attributes are required. Attributes associated with a tag may be specified in any order.

A tag also may have one or more elements associated with it. An element consists of an associated tag or a text string. If a tag has an associated text string, the string can immediately follow the period (.) delimiter of the tag, or it can be put at the beginning of the line immediately following the tag.

Some tags have end tags associated with them. The markup for an end tag is **:e** immediately followed by the tag name, followed by a period (.). In the case where a tag has an associated end tag, its markup tells the computer, "Start doing my thing until you see my end tag." If you forget to include an end tag, the compiler tells you by displaying an error message.

The best way to learn about tags is to look at the examples provided in the following sections and then create some examples of your own. The more you experiment with tags, the more your knowledge of their characteristics grows.

Defining a Panel with IPF Markup

A panel is the text unit you have written to describe a window, an entry field, or a selection choice of the application interface. The panel is displayed in the help text window when a user requests help for the item it describes.

Headings — The :h Tags

The beginning of a panel is defined by an **:h** tag. Every panel has a title that is displayed in the help text window. The title usually is the same as that of the item you are describing.

For Example: If you are describing a selection choice entitled "Copy Program," you might enter the first line of your panel like this:

```
:h1 res=001 Copy Program
```

There are six levels of headings provided by IPF. In our example, we have used the topmost level, the **:h1** tag. If you are planning on using more than one heading level, you may want to refer to "Source File Structure" on page 17-4.

The **res=** attribute shown in the example specifies the panel identifier, which is an integer IPF uses to associate the panel with the item for which help is requested. In this case, 001 is the identifier for the **Copy Program** selection choice. Every panel *must* have a panel identifier, and it must be an integer. IPF allows integers 1 through 32 000.

The "Copy Program" text string is put in the title bar of the help text window. It also appears as an entry in the contents window. If you do not want a title to appear as an entry in the contents window, you can specify the **hide** attribute.

For Example: To prevent your help title from appearing in the contents window, you would tag it like this:

```
:h1 res=001 hide.Copy Program
```

The text string for the title also may be typed at the beginning of the line immediately following the **:h1** tag.

For Example: You could have entered your heading like this:

```
:h1 res=001 hide.
Copy Program
```

This is true for all tags that have text strings following the period.

The **:h1** tag defines the beginning of the "Copy Program" panel. The end of a panel is defined by the next :h1 tag or by end-of-file.

The order of panels in the file determines the order of entries in the contents window. The heading of the first panel in the file would be the first entry, and so on.

Index Entries

When the following index tags are used in panels, IPF arranges the entries associated with the tags in alphabetic order in list box windows. The windows these entries are associated with, are mentioned in their descriptions.

The :i1. and :i2. Tags

You should create at least one index entry for each panel, using the **:i1** tag.

For Example: If you wanted an index entry in the help index window that looked like this:

copy program

you would tag it this way:

```
:i1.copy program
```

You may decide to create two levels of index entries, using both the **:i1** and the **:i2** tags. This is done with the **id** attribute of the **:i1** tag and the **refid** attribute of the **:i2** tag.

For Example: If you create the following index entries, they are in your source file in the sequence shown, but they are located in different panels.

```
:i1 id=prnt.printers and plotters
:i2 refid=prnt.change printer
:i2 refid=prnt.add printer
:i2 refid=prnt.printer properties
```

When the user selects the **Help index** choice from the help pull-down, the index entries would look like this:

```
printers and plotters
  add printer
  change printer
  printer properties
```

If an **:i1** tag has an **id** that is referred to by the **refid** attribute of the **:i2** tag, the **:i1** tag must precede the **:i2** tags in the file.

One way to keep track of **:i1** tags that are referred to by **:i2** tags is to keep them in a separate file. You can then imbed the file when it is time to compile your source file. For more information, see "Source File Structure" on page 17-4.

Synonyms — The :isyn. Tag

The **:i1** tag has a **roots=' '** attribute that can be associated with the **:isyn.** tag. The purpose of this association is to enhance a user search request by including index entries that have been identified as synonyms in the results of an index search.

For Example: You have the following entries in a panel:

```
:isyn root=copy.
copy copying duplicate duplicating
:isyn root=folder.
folder folders document documents
:i1 roots='copy folder'.
copying a document
```

The **roots=' '** attribute of the **:i1** tag associates the entry with the the **root** attributes of two **:isyn** tags.

If a user search request of the index includes one of the words in either of the two synonym text strings of the **:isyn** entries, the search results will include not only **:i1** entries that contain the word, but also any **:i1** entries like the one shown above that have been associated with the word.

For Example: A user enters the word 'duplicating' in a search request. When the search is completed,

copying a document

appears as one of the entries in the search results window.

See "Services Action Bar Pull-Down" on page 15-3 for a description of the **Search** option.

Commands — The :icmd. Tag

Panels for application commands can be associated with a command entry field by imbedding the **:icmd.** tag with a command name in the panel that describes the command.

When the cursor is positioned in the associated entry field and the user presses F1 or selects the **Help** pushbutton, titles of panels that contain these tags are displayed in alphabetic order in a list box window.

If the user types a command name in the entry field and then requests help, the panel associated with the command is displayed.

Note: The association with the entry field and command names is a programming task. In addition, the application developer must define the field with which command panels are to be associated as a command entry field. For more information, see "Defining a Command Entry Field" on page 16-12.

Source File Structure

Following are some of the elements which need to be in a source file. Figure 17-2 on page 17-5 illustrates an IPF source file.

The :userdoc. Tag

The **:userdoc.** tag is required as the first line of an IPF source file. This tag is a signal to the IPF Compiler (IPFC) to begin translating the markup that follows. The **:euserdoc.** tag is required as the last line of the file to signal the end of the tagged document.

Heading Sequences

In "Defining a Panel with IPF Markup" on page 17-2, some aspects of heading tags in relation to panel definition are described. How the sequence of panels in the file determine the order of entries in the contents window and how a tree-structured contents can be created is also described.

If you want to organize your information by topics and subtopics, so that a tree-structured contents is created, you can accomplish this by using descending heading sequences to define panels. Heading sequences must always start with a **:h1.**, and each sequence must not skip a level in the heading hierarchy. For example, you cannot use only **:h2** and **:h3** tags, and you cannot use only **:h1** and **:h3** tags.

See "Options Action Bar Pull-Down" on page 15-5 for a description of the display of a tree-structured contents.

```
:userdoc.
:h1 res=001.Help for Print Manager
.im INDEX.IPF
.* Imbed file of i1 index tags referred to by i2 index tags
:i1.print manager
:i2 refid=prnt.add to queue
:p.
The Print Manager controls printing and plotting.
When you ask that a document be printed, the spooler
creates a print job and adds it to the appropriate queue.
If the spooler is not enabled, printing is immediate.
The print jobs go directly to the printer and might be combined
with other jobs.
:
:euserdoc.
```

Figure 17-2. Source File Structure

The default heading levels that will appear in the contents window are **:h1** through **:h3**. If you decide to use any of the **:h4** through **:h6** tags, these tags do not define separate panels. Instead, the headings specified with these tags will appear as part of the text of the panel when it is displayed.

Imbedded Files — .im

The **.im** control word can be used to imbed other source files in a base source file.

For Example: In Figure 17-2,

.im INDEX.IPF

imbeds a source file containing index entries.

If you are imbedding files, the source file that begins with the **:userdoc.** tag is considered the base file.

If a file to be imbedded in the base file contains panel definitions, it can be imbedded anywhere in the base file preceding or following a panel definition or an **:fn.** tag. Of course, the placement of an imbed is determined by the order in which you want your contents entries to appear.

A file containing index entries cannot be imbedded preceding the first panel definition or between **:fn.** and **:efn** tags. Notice in Figure 17-2 that this file is imbedded immediately after the first **:h** tag. Because the file contains only index entries, this placement is satisfactory. However, if the file contained additional panels, you would want to imbed the file preceding the start of the panel.

Imbedded files must not use the **:userdoc.** and **:euserdoc.** tags, because at the time the file is imbedded the compiler has already been signaled to start its translation.

Naming Source Files

It is a good idea to give your source files an extension of IPF, so they can be distinguished from your other files. IPFC does not require an IPF extension; however, if two files have the same name, and one of them has an IPF extension, IPFC reads the one with the IPF extension.

For Example: Here is the name of a source file:

MYHELP.IPF

Commenting Files — .*

The .* control word allows you to place lines of comments in your file. Any text on the same line as the control word is ignored by the compiler.

For Example: The following lines would be recognized as comment and would be ignored by the compiler:

```
.*****************************
.* Panel IDs 100 through 300
.* are contained in this file.
.*****************************
```

Text Presentation

Once your panel is defined, you may want to make the text look more attractive when it is displayed in the help text window. This section explains how you can use tags to:

- Highlight text.
- Add notes, notices, and lists.
- Illustrate your text with examples, figures, and character graphics.
- Control the formatting of lines of text.
- Create special characters that do not appear on the keyboard.

Highlighting Text — The :hp. Tags

Use the **:hp.** (highlighted phrase) and **:ehp.** (end highlighted phrase) tags to emphasize text by setting it in a different font or color. IPF provides nine pairs of **:hp.** tags (**:hp1.** to **:hp9.**) for nine styles of highlighting. The tags are paired because you need one tag to turn on the highlighting and its pair to turn off the highlighting.

You can place **:hp.** tags anywhere in a line of text. The colon does not have to be in the first column.

For Example: If you wanted to make a word bold, like this:

Read **this** first.

you would tag it this way:

Read :hp2.this:ehp2. first.

The highlighting produced by the nine sets of tags is as follows:

:hp1.	*Italics*
:hp2.	**Bold**
:hp3.	***Bold Italics***
:hp4.	Color 1 (Red)
:hp5.	Underline
:hp6.	*Underline italics*
:hp7.	**Underline bold**
:hp8.	Color 2 (Blue)
:hp9.	Color 3 (Pink)

Colors may vary depending on the type of display you use.

Notes

If you want to include notes in your help information, IPF has two tags you can use, the **:note.** tag and the **:nt.** tag with its corresponding **:ent.** tag. Which one you use depends on whether your note will contain more than one paragraph.

The :note. Tag for Notes

Use the **:note.** tag for one-paragraph notes. You do not need a tag to end the note because the system ends it when it encounters the next tag.

For Example: If you wanted to create a note like this:

Note: Fill in all entry fields before leaving this panel.

you would tag it this way:

:note. Fill in all entry fields before leaving this panel.

Your notes do not have to be preceded by the word "Note." The **:note.** tag has an attribute, **text=' '.** that allows you to label your notes any way you want.

For Example: If you wanted to create a note like this:

Reminder: Fill in all entry fields before leaving this panel.

you would tag it this way:

:note text='Reminder'. Fill in all entry fields before leaving this panel.

The :nt. Tag for Notes

Use the **:nt.** tag to create notes with more than one paragraph. Remember to end the note with an **:ent.** tag.

For Example: If you wanted to create a note like this:

Note: Fill in all entry fields before leaving this panel. If you do not, all your information will be lost.

If your information is lost, retype it in the entry fields.

you would tag it this way:

```
:nt. Fill in all entry fields before leaving this panel.
If you do not, all your information will be lost.
:p.If your information is lost, retype it in the entry
fields.
:ent.
```

You can include many paragraphs in a note.

Notices — The :caution. and :warning. Tags

Two sets of tags allow you to include caution and warning notices in your panels.

Use the **:caution.** tag to alert users to a risk of possible damage to applications or data. Use the **:warning.** tag to alert users to a more severe risk or possible error condition in the system. In both cases, place the caution and warning statements before the help information to which they apply so the user is cautioned or warned in advance. Both tags require end tags. Also, you may use the **text=' '.** attribute if you want to use words other than "Caution" and "Warning" with these notices.

For Example: If you wanted to create a caution notice like this:

CAUTION:
Your data will be lost if you do not do something.

you would tag it this way:

```
:caution.
Your data will be lost if you do not do something.
:ecaution.
```

For Example: If you wanted to create a warning notice like this:

Warning: The disk contains bad sectors.

you would tag it this way:

```
:warning.
The disk contains bad sectors.
:ewarning.
```

Lists

One common way of presenting information in panels is in list form. IPF provides tags to create five kinds of list:

- Simple
- Unordered
- Ordered
- Definition
- Parameter.

All five lists are created in much the same way. A tag begins the list, another tag identifies each item in the list, and another tag ends the list.

The :sl. Tag for Simple Lists

Simple lists are vertical arrangements of items without any kind of character, symbol, letter, or number preceding the items in the list. Use simple lists when the order of the items is not important and when there is no need to set off each list item with some kind of symbol or character.

To create a simple list, use the **:sl.** (for simple list) tag to begin the list and its corresponding **:esl.** (for end simple list) tag to end the list. Identify each item in the list with an **:li.** (for list item) tag.

For Example: If you wanted to create a simple list like this:

File
Edit
View
Options

you would tag it this way:

```
:sl.
:li.File
:li.Edit
:li.View
:li.Options
:esl.
```

The **:sl.** tag has an attribute, **compact**, which causes the list to format without blank lines between the list items.

For Example: If you wanted to create a compact simple list like this:

File
Edit
View
Options

you would tag it this way:

```
:sl compact.
:li.File
:li.Edit
:li.View
:li.Options
:esl.
```

The :ul. Tag for Unordered Lists

Unordered lists are vertical arrangements of items with each item in the list preceded by a typographical character. The two typographical characters used are the *bullet* (•) and the *hyphen* (-). Hyphens appear in *nested lists*, that is, lists within lists. Nested lists will be explained on page 17-9.

Use unordered lists when the order of the items is not important but you need to set off each list item with a bullet. It is a good idea to use an unordered list when the list items are longer than one line. If list items are short use a simple list.

To create an unordered list, use the **:ul.** (for unordered list) tag to begin the list and its corresponding **:eul.** (for end unordered list) tag to end the list. Identify each item in the list with an **:li.** tag.

For Example: If you wanted to create an unordered list like this:

- Information typed in Panel A will be stored in the STORES.DAT file in whatever directory you designate.
- Information typed in Panel B will be stored in the SALES.DAT file in the current directory.
- Information typed in Panel C will be stored in the LOSSES.DAT file in the C:\FINANCE directory.

you would tag it this way:

```
:ul.
:li.Information typed in Panel A will be stored in
the STORES.DAT file in whatever directory you
designate.
:li.Information typed in Panel B will be stored in
the SALES.DAT file in the current directory.
:li.Information typed in Panel C will be stored in
the LOSSES.DAT file in the C:\FINANCE direc-
tory.
:eul.
```

The **:ul.** tag has an attribute, **compact**, which causes the list to format without blank lines between the list items.

For Example: If you wanted to create a compact unordered list this:

- Information typed in Panel A will be stored in the STORES.DAT file in whatever directory you designate.
- Information typed in Panel B will be stored in the SALES.DAT file in the current directory.
- Information typed in Panel C will be stored in the LOSSES.DAT file in the C:\FINANCE directory.

you would tag it this way:

```
:ul compact.
:li.Information typed in Panel A will be stored in
the STORES.DAT file in whatever directory you
designate.
:li.Information typed in Panel B will be stored in
the SALES.DAT file in the current directory.
:li.Information typed in Panel C will be stored in
the LOSSES.DAT file in the C:\FINANCE direc-
tory.
:eul.
```

The :ol. Tag for Ordered Lists

Ordered lists are vertical arrangements of items with each item in the list preceded by a number or letter. Letters appear in *nested lists*, that is, lists within lists. Nested lists will be explained in the next section.

Use ordered lists when the order of the items is important, such as when describing a procedure whose steps have to be followed in order.

To create an ordered, use the **:ol.** (for ordered list) tag to begin the list and its corresponding **:eol.** (for end ordered list) tag to end the list. Identify each item in the list with an **:li.** tag.

For Example: If you wanted to create an ordered list like this:

1. Open the diskette door drive.
2. Remove the diskette.
3. Store the diskette in a safe place.

you would tag it this way:

```
:ol.
:li.Open the diskette door drive.
:li.Remove the diskette.
:li.Store the diskette in a safe place.
:eol.
```

The **:ol.** tag has an attribute, **compact**, which causes the list to format without blank lines between the list items.

For Example: If you wanted to create a compact ordered list this:

1. Open the diskette door drive.
2. Remove the diskette.
3. Store the diskette in a safe place.

you would tag it this way:

```
:ol compact.
:li.Open the diskette door drive.
:li.Remove the diskette.
:li.Store the diskette in a safe place.
:eol.
```

Nesting Lists

Sometimes an item in your simple, unordered, and ordered lists my need to be explained further with a list of its own. This is called a *nested list*, a list contained within another list. Here is an example of nested, unordered lists, used to describe a directory structure:

- C:\REPORTS\SALES.87
 - FIRST.QTR
 - SECOND.QTR
 - THIRD.QTR
 - FOURTH.QTR
- C:\REPORTS\SALES.88
 - FIRST.QTR
 - SECOND.QTR
 - THIRD.QTR
 - FOURTH.QTR
- C:\REPORTS\SALES.89
 - FIRST.QTR
 - SECOND.QTR
 - THIRD.QTR
 - FOURTH.QTR

Notice that a • (bullet) precedes items in the first-level list and that a – (dash) precedes items in the second-level lists. The bullets and hyphens alternate for each level of list. That is, third-level list items would be preceded by bullets, fourth-level by hyphens, and so forth. The same thing happens with ordered lists. The list items in the first-level list are numbered, the items in the second-level list are lettered, the items in the third level list are numbered, and so forth.

Here is how you would tag the example above:

```
:ul compact.
:li.C:\REPORTS\SALES.87
:ul compact.
:li.FIRST.QTR
:li.SECOND.QTR
:li.THIRD.QTR
:li.FOURTH.QTR
:eul.
:li.C:\REPORTS\SALES.88
:ul compact.
:li.FIRST.QTR
:li.SECOND.QTR
:li.THIRD.QTR
:li.FOURTH.QTR
:eul.
:li.C:\REPORTS\SALES.89
:ul compact.
:li.FIRST.QTR
:li.SECOND.QTR
:li.THIRD.QTR
:li.FOURTH.QTR
:eul.
:eul.
```

When nesting lists, make sure you end each list with an end-list tag. For every **:sl.**, **:ul.**, and **:ol.** there must be a corresponding **:esl.**, **:eul.**, and **:eol.**.

The :dl. Tag for Definition Lists

A definition list is a special list that pairs a term with a description of the term.

To create a definition list, use the **:dl.** (for definition list) tag to begin the list and its corresponding **:edl.** (for end definition list) tag to end the list. Identify each term in the list with a **:dt.** (for definition term) tag and each definition with a **:dd.** (for definition description) tag.

For Example: If you wanted to create a definition list like this:

Insert key Switches between insert and replace modes.

Home key Moves the cursor to the beginning of the current line.

End key Moves the cursor to the end of the current line.

you would tag it this way:

```
:dl.
:dt.Insert key
:dd.Switches between insert and replace modes.
:dt.Home key
:dd.Moves the cursor to the beginning of the
current line.
:dt.End key
:dd.Moves the cursor to the end of the current line.
:edl.
```

You may want to place headings over the columns of terms and definitions. To do this, use the **:dthd.** (for definition term heading) tag to identify the heading over the definition terms and the **:ddhd.** (for definition description heading) tag to identify the heading over the definition descriptions.

The **:dl.** tag has several attributes that let you control the formatting of definition lists:

- The **tsize=***character spaces* attribute specifies the space for the definition list terms and term heading. The default is 10.
- The **compact** attribute formats the list without a blank line between items.
- The **break=** attribute designates where the definition descriptions are supposed to start; **break=none** places the description on the same line as the term. This is the default. **break=fit** places the description on the line below the term if the term is longer than the **tsize** value. **break=all** places the description on the line below the term.

For Example: If you wanted to create a definition list similar to the one above, except compact, with headings, and with the definition descriptions on the lines below the terms so that it looked like this:

Key Purpose

Insert key
Switches between insert and replace modes.

Home key
Moves the cursor to the beginning of the current line.

End key
Moves the cursor to the end of the current line.

you would tag it this way:

```
:dl compact break=all.
:dthd.Key
:ddhd.Purpose
:dt.Insert key
:dd.Switches between insert and replace modes.
:dt.Home key
:dd.Moves the cursor to the beginning of the
current line.
:dt.End key
:dd.Moves the cursor to the end of the current line.
:edl.
```

The :parml. Tag for Parameter Lists

Parameter lists are similar to definition lists in appearance and the way you use tags to create them. A parameter list is a two-column list like a definition list. The differences are that parameter lists cannot have headings.

Parameter list tags are similar to definition list tags. The **:parml.** (for parameter list) tag begins the list and its corresponding **:eparml.** (for end parameter list) tag ends the list. Terms in the list are identified by a **:pt.** (for parameter term) tag and descriptions are identified with a **:pd.** (for parameter description) tag.

The **:parml.** tag uses the same attributes as the **:dl.** tag, described above.

For Example: If you wanted to create a parameter list like this:

Insert key
Switches between insert and replace modes.

Home key
Moves the cursor to the beginning of the current line.

End key
Moves the cursor to the end of the current line.

you would tag it this way:

```
:parml compact break=all.
:pt.Insert key
```

:pd.Switches between insert and replace modes.
:pt.Home key
:pd.Moves the cursor to the beginning of the current line.
:pt.End key
:pd.Moves the cursor to the end of the current line.
:eparml.

Illustrating Text

You may want to illustrate your help information with textual examples or simple line drawings created with ASCII characters. Three sets of IPF tags allow you to place examples, figures, and character graphics in your help information.

The :xmp. Tag for Examples

One way of making help information more understandable is to provide users with examples. The **:xmp.** (for example) tag and its corresponding **:exmp.** (for end example) tag allow you to illustrate your information with textual examples that appear in a monospaced font, just as you have typed them. That is, the **:xmp.** tag turns off formatting so that you can arrange text any way you want it without having the compiler reformat it.

For Example: If you wanted to create an example like this:

```
File   Edit   View   Options   Help

               All
               Some . . .
               By . . .
```

you would tag it this way:

```
:xmp.
File   Edit   View   Options   Help

               All
               Some . . .
               By . . .
:exmp.
```

The :fig. and :figcap. Tags for Figures and Captions

The **:xmp.** tag gives you one way to to illustrate your information with textual examples. The **:fig.** (for figure) tag and its corresponding **:efig.** (for end figure) tag give you another way. And the **:figcap.** tag allows you to caption illustrations.

Like the **:xmp.** tag, the **:fig.** tag turns off formatting so that you can arrange text without having the compiler reformat it. Textual examples are displayed almost as you type them; "almost" because the system uses a proportional font after the **:fig.** tag, meaning that not all letters take up the same amount of space. For example, a capital W will be wider than a lowercase i. As a result, some text may not be displayed exactly as you align it when you type it. You will have to readjust the alignment by adding or deleting blanks.

Use the **:figcap.** tag to place a caption under your figures. Type the tag immediately after the **:fig.** tag or immediately before the **:efig.** tag.

For Example: If you wanted to create a figure like this:

```
Files                Directories

FREDPLAN.TXT          ..
MARGPLAN.TXT         PROJECTX
GLADPLAN.TXT          [A:]
GLADSKED.TXT          [B:]
```

Lists of Files and Directories

you would tag it this way:

```
:fig.
Files                Directories

FREDPLAN.TXT          ..
MARGPLAN.TXT         PROJECTX
GLADPLAN.TXT          [A:]
GLADSKED.TXT          [B:]

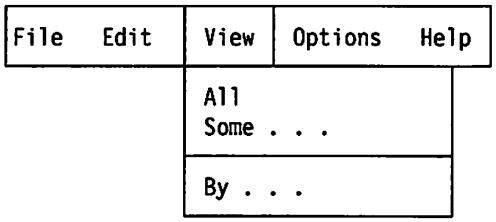
:figcap.Lists of Files and Directories
:efig.
```

The :cgraphic. Tag for Character Graphics

If you want to include line drawings created with ASCII characters use the **:cgraphic.** (for character graphics) and **:ecgraphic.** (for end character graphics) tags to do this.

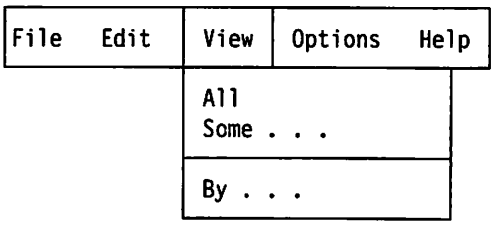
The tags are placed before and after the character graphic.

For Example: If you wanted to create an example like this:



you would tag it this way:

```
:cgraphic.
```




```
:ecgraphic.
```

Controlling Line Formatting

You can control the formatting of text with the **:lines.** tag.

The :lines. Tag for Controlling Formatting

Use the **:lines.** tag to turn off formatting by the compiler so that text typed after the tag is displayed almost as you typed it; "almost" because the system uses a proportional font after the **:lines.** tag, meaning that not all letters take up the same amount of space. For example, a capital W will be wider than a lowercase i. As a result, some text may not be displayed exactly as you align it when you type it. You will have to readjust the alignment by adding or deleting blanks.

Use the **:elines.** tag to turn formatting back on.

For Example: If you wanted to create text like this:

```
******************************
Your data will be LOST
        if
    you do not
   DO SOMETHING
******************************
```

you would tag it this way:

```
:lines.
******************************
Your data will be LOST
        if
    you do not
   DO SOMETHING
******************************
:elines.
```

Creating Special Characters

Your keyboard probably does not contain all the characters you may want to use in your help information. IPF provides a set of tags that create special characters and symbols like:

Å — created with the **&Ao.** (for A overcircle) tag

↓ — created with the **&darrow.** (for down arrow) tag

└ — created with the **&bxll.** (for box lower-left) tag

Notice that all tags begin with an ampersand (&) and end with a period (.). Look under the "Symbols" heading in Appendix H, "Information Presentation Facility Tags" on page I-1 for the list of tags you can use.

You can place symbols tags anywhere in a line of text. The ampersand does not have to be in the first column.

Graphic Illustration

Sometimes you may want to include illustrations which cannot be created using character graphics. These can be created as bit maps and are included in your source file by using the **:artwork.**tag.

The :artwork. Tag

The **:artwork.** tag allows you to illustrate your text with a graphic, called a bit map. The bit map resides in a file that is loaded when your text is displayed. The bit map can be any one of the following types, created on external media, such as:

Vector graphics
Scanned images
Graphic images.

The bit map also can be one the application creates or one that was created with the Presentation Manager Icon Editor. The **name=' '** attribute specifies the name of the bit map file.

The **align=** attribute allows you to position the graphic to the left, right or center of the current margin.

The graphic can be placed within a line of text by using the **runin** attribute.

For Example: If you want to include an icon within a line of text, the tags would be as follows:

```
The archive flag is set to
:artwork runin name='ONGRAF.BMP'.
when you create or change a file.
```

To make your graphics more useful, you can include **:link.** tags, to make graphic areas selectable. For information on how to do this, see "Linking from Hypergraphic Areas" on page 17-16.

Linking from Hypertext Phrases — The :link. Tag

The capability for linking online information so the user can navigate directly from a selectable phrase in the context of a panel to information related to the phrase is referred to as hypertext. The **:link** tag enables you to make a phrase in a panel selectable. When a hypertext phrase is selected, the link can:

- Display another panel in the text window
- Display a smaller window within the text window
- Send a message to the application
- Start another Presentation Manager application.

IPF allows as many hypertext phrases within a panel as you want to include.

One advantage of using the **:link** tag to link the user to explanations of phrases is that the same information can be linked to from different panels. Another advantage is that the user, who may not need the explanation, makes the decision to access it.

Linking to Another Panel

You use the **reftype=hd** attribute with **:link** when you want to link the user to another panel.

For Example: If the text of a help panel contains the following:

```
:p.
Use this choice to change the
:link reftype=hd res=2003.
flags
:elink.
that are assigned to the
selected file or files.
```

The **hd** keyword indicates the link is to another panel. The **res=' '** attribute identifies the ID of the panel being linked to.

Note: The default for heading tags that define panels is **:h1.**, **:h2.**, and **:h3.** tags. Headings with **:h4.**, **:h5.**, and **:h6.** tags appear within the text of the panel and cannot be linked to.

When the panel for the text in the example is viewed, the fact that the hypertext phrase "flags" can be selected is indicated to the user by a unique highlighting. If the user selects the phrase, the panel containing the explanation for the phrase appears in the text window.

The above example is part of Figure 17-4 on page 17-15. The source for the explanation panel related to "flags" is also included in this figure.

Note: The highlighting of a hypertext phrase is done with a color selected by IPF and should not be confused with highlighted phrase tags, which are used to change the type font. (See "Highlighting Text — The :hp. Tags" on page 17-6 for an explanation of these tags.)

The panel source shown in Figure 17-5 on page 17-16 contains several examples of the **:link** tag used for list items in an unordered list. When this source is compiled and is viewed in the help text window, it looks like the window shown in Figure 17-3 on page 17-14. The user may select any of the reference phrases in the list:

```
system time
system date
cursor blink rate
mouse double-click rate.
```

If the user selects a phrase, another panel replaces the one shown in the text window.

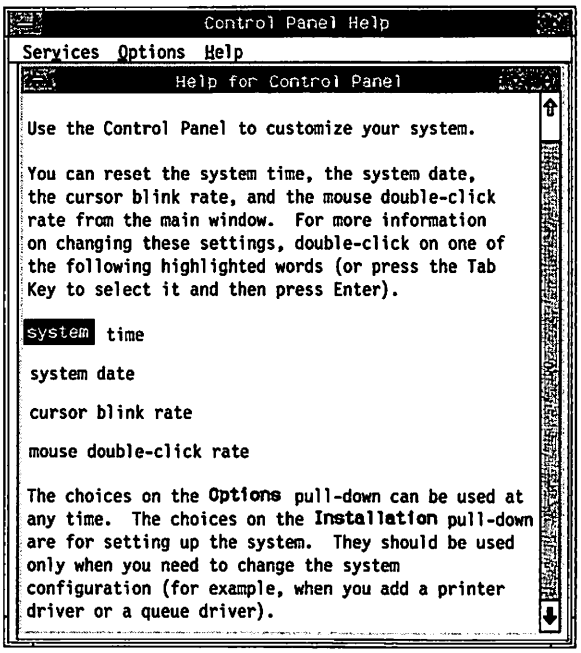


Figure 17-3. Help for Control Panel Example

Notice the different tagging methods used in Figure 17-4 on page 17-15 and Figure 17-5 on page 17-16. The writer who created the source shown in Figure 17-5 on page 17-16 preferred to enter the elements of the **:link** tag all on one line. Since there are multiple similar links, the attributes of the four link tags line up in neat columns, making it easy to check tag syntax.

The writer who created the source shown in Figure 17-4 on page 17-15 preferred to put each element of the single **:link** on a separate line.

Both methods are correct.

Linking to a Smaller Window

You use the **reftype=fn** attribute with **:link** when you want to link the user to additional text contained in a smaller window. The **fn** keyword refers to a **:fn** tag.

Example 1: If the value of the footnote ID is "systime," the link tag might look like this:

```
:link refid=systime reftype=fn.
system time
:elink.
```

The **refid** attribute of the **:link** tag specifies the value of the footnote ID shown in Example 2 on page 17-14.

The :fn. Tag

The text contained within **:fn** and **:efn** tags will never be seen by users unless you link them to it with the **:link** tag. Therefore, the footnote has an **id** attribute but does not require a **res** attribute because it is not associated with an entry field or an application window; instead, it is associated with a panel.

Example 2: If this is the footnote referenced in Example 1.

```
:fn id=systime.
Here is the explanation of system time
referenced by the link to a footnote
shown in Example 1.
:efn.
```

When the hypertext phrase "system time" in Example 1 is selected, the text in the footnote is displayed in a smaller window.

A footnote does not cause an entry in the contents window. A footnote cannot contain index entries. The title of the smaller window that appears when a reference phrase to a footnote is selected is the text of the reference phrase. For example, the title of the footnote window in which the above footnote would be displayed would be the hypertext phrase "system time." from the previous example.

A footnote can be placed anywhere in your source file, as long as it follows the first **:h** tag and is at the end of the text for a panel. For more information on file structure, see "Source File Structure" on page 17-4.

Sending a Message to the Application

When the **reftype=inform** attribute is specified with the **:link.** tag, a message is sent to the application. The **res=** attribute is used in a different way here. Instead of being a resource identifier for IPF (a panel ID), it is a resource identifier for the application. The value specified must be an integer. When the application receives the message, it may then perform an application-specific function.

```
.*********************************************************************
:h1 res=019.
Help for Change flags
:i2 refid=file.change flags
:p.
Use this choice to change the
:link reftype=hd res=2003.
flags
:elink.
that are assigned to the selected file or files.
:p.
You can change the archive, read-only, hidden, and system flags of
one or more files.
.*********************************************************************
:h1 res=2003.
Help for Flags
:i2 refid=file.Flags
:p.
Flags are the characteristics of your files and directories.
The system uses flags to identify the kind of file or directory.
:p.
Four types of flags are available:
:dl tsize=5 break=all.
:dt.
System
:dd.
Indicates that the file or the directory is part of the system.
:dt.
Hidden
:dd.
Indicates that the file or the directory should not be displayed in the
Directory Tree window.
:dt.
Read-only
:dd.
Indicates that the file can be read but it cannot be changed.
:dt.
Archive
:dd.
Indicates how the BACKUP, RESTORE, COPY, and XCOPY commands work
on the file.
:edl.
:p.
The archive flag is set to on when you create or change a file.
The commands BACKUP, RESTORE, COPY, and XCOPY look for this flag,
and they work only on files whose flag is set to on.
If you want these commands to work even on files that have not been
changed, you must set the archive flag to on.
.*********************************************************************
```

Figure 17-4. A Link to Another Panel. This example is part of the source file of help responses for the *File Manager* application.

Programming Note: The message sent to the application is HM_INFORM.

```
:h1 res=001.Help for Control Panel
:i1 id=cp.control Panel
:p.
Use the Control Panel to customize your system.
:p.
You can reset the system time, the system date, the cursor blink
rate, and the mouse double-click rate from the main window.
For more information on changing these settings, double-click on
one of the following highlighted words (or press the Tab key to
select it and then press Enter).
:sl.
:li.:link res=2001 reftype=hd.system time:elink.
:li.:link res=2002 reftype=hd.system date:elink.
:li.:link res=2003 reftype=hd.cursor blink rate:elink.
:li.:link res=2004 reftype=hd.mouse double-click rate:elink.
:esl.
:p.
The choices on the :hp2.Options:ehp2. pull-down can be used at any time.
The choices on the :hp2.Installation:ehp2. pull-down are
for setting up the system.
They should be used only when you need to change the system configuration
(for example, when you add a printer driver or a queue driver).
:p.
Use the :hp2.Help:ehp2. action bar choice to look at how to use help.
You can also use help to get extended help, to go to a list of keys, to
go to the help index, or to go to the table of contents.
:p.
To close the Control Panel, either:
:ul.
:li.Select the :hp2.Exit:ehp2. choice on the :hp2.Options:ehp2. pull-down
:li.Select the :hp2.Close:ehp2. choice on the system menu pull-down
:li.Press F3
:li.Press Alt+F4
:li.Double-click on the system menu.
:eul.
```

Figure 17-5. Source for a Panel with Multiple Links. This example is excerpted from the source file of help responses for the *Control Panel* application.

Starting Another Presentation Manager Application

The **reftype=launch** attribute of the **:link.** tag causes IPF to start another Presentation Manager application. The **object=** attribute specifies the fully qualified file name of the application. The **data=** attribute specifies parameters associated with the application to be started.

This attribute can be used with the **:link.** tag to start a tutorial.

Linking from Hypergraphic Areas

Now that you know how the **:link.** tag makes a text phrase within a panel selectable, and you also know what can happen when the hypertext phrase is selected, you might be interested to know that you also can make a graphic selectable. A selectable graphic is referred to as a hypergraphic area. When a hypergraphic area is selected, the link can:

- Display another panel in the text window
- Display a smaller window within the text window
- Send a message to the application

- Start another Presentation Manager application.

In "Graphic Illustration" on page 17-12, the **:artwork.** tag, which is used to include graphics in text panels, is discussed. The **:artwork.** tag has an attribute **linkfile=' '** which lets you make the graphic selectable.

Link Definition File — The :artlink. Tag

A link definition file must start with the **:artlink.** tag and end with its end tag, **:eartlink.**. These tags identify the file to the compiler as a file that defines the selectable area (or areas) of the graphic identified by the **name=' '** attribute of the **:artwork.** tag. The **name=' '**attribute specifies the fully-qualified name of the file containing the bit map.

You can define the entire bit map as one selectable area and have one link to other information. In this case, your link definition file contains only one link definition.

For Example: If you are including an icon in a line of text. You want to make the icon a hypergraphic area. When the user selects the icon, you want to display a panel of information associated with the icon. The contents of your link definition file might look like this:

```
:artlink.
:link res=999 reftype=hd.
:eartlink.
```

Notice that a **:link.** tag in a link definition file does not have a matching **:elink.** tag.

If the name of your link definition file is MYLINK.ART, and the name of the file containing the bit map is MYICON.BMP, your **:artwork.** tag might look like Example 1 in Figure 17-6 on page 17-18.

You also can define two or more segments of the bit map as selectable and link to different information for each of the selectable areas.

Example 2 in Figure 17-6 on page 17-18 shows what the link definition file might look like for a bit map that has been defined as having three selectable areas. The x, y, cx, and cy values define the hypergraphic areas. When the user selects one of the hypergraphic areas defined in the link definition file shown in Example 2, a smaller window is displayed that contains information that is specific to the hypergraphic area.

Compiling Source Files

When OS/2 1.2 Programming Tools is first installed, the following environment variable is placed in the CONFIG.SYS file:

```
IPFC=C:\TOOLKIT12\IPFC
```

The IPFC environment variable identifies the directory in which data files needed by IPFC are stored.

Following are the parameters you can specify with the IPFC command at the OS/2 command prompt, to have the IPF Compiler compile your .IPF source file.

Format

IPFC *file name*.IPF [[/COUNTRY] [/CODEPAGE][/LANGUAGE][/X][/W]]

Parameters

/COUNTRY=*nnn*
: The *nnn* is the 3-digit country code. The country code determines the collating sequence. See Table 17-2 on page 17-19.

/CODEPAGE=*nnn*
: The *nnn* is the 3-digit code page. See Table 17-2 on page 17-19.

/LANGUAGE=*xxx*
: The *xxx* is a 3-letter identifier that indicates the NLS file to be used. The NLS file determines the language that is displayed for the **Note**, **Warning**, and **CAUTION** titles of the **:nt.**, **:note.**, **:warning.**, and **:caution.** tags.

 The NLS file also determines the characters displayed for ordered list numbers and unordered list dingbats. See Table 17-1 on page 17-19.

/X
: Generates and displays a cross-reference list.

/W*n*
: Generates and displays a list of error messages. The *n* indicates the level of error messages you want to receive. Values you can specify for *n* are 1, 2, or 3. See "Error Message File" on page 17-18.

Example 1: An :artwork. Tag

:artwork name='C:\MYICON.BMP' runin linkfile='C:\MYLINK.ART'.

Note: File names must be fully qualified and enclosed in single quotes.

Example 2: A Link Definition File for a Segmented Bit Map

```
:artlink.
:link reftype=fn refid=ab01 x=0 y=0 cx=16 cy=8 .
:link reftype=fn refid=ab02 x=16 y=8 cx=16 cy=8.
:link reftype=fn refid=ab03 x=0 y=8 cx=16 cy=8.
:eartlink.
```

Note: Link definitions for hypergraphic areas also can include linking to another panel, sending a message to the application, or starting another Presentation Manager application.

Figure 17-6. Hypergraphic Links

Note: The abbreviated forms recognized by IPFC for /COUNTRY, /CODEPAGE, and /LANGUAGE are, respectively; /COU, /COD, and /L.

Dialog Manager Support

For information concerning Dialog Manager support, see the Dialog Manager publications in Programming Tools and Information.

National Language Support (NLS)

The following parameters provide NLS country support:

/COUNTRY
/CODEPAGE
/LANGUAGE

If you do not specify these parameters, the default for the first two are the values specified in your CONFIG.SYS file. If you do not request an NLS language file be used, the titles for the tags listed in the parameter description are shown in U.S. English.

Error Message File

The /W*n* parameter of IPFC determines the levels of error messages that will be displayed. Following are the meanings of the values you can specify for *n*:

Value	Meaning
1	Returns only warning level 1 messages. Warning level 1 messages are the most severe.
2	Returns warning level 1 and 2 messages.
3	Returns all three warning levels of messages. This is the default. (Warning level 3 messages are the least severe.)

When IPF compiles your file, it generates and displays the error messages. If no errors are found, IPF will tell you that compiling has been completed and no errors were found.

You may prefer to redirect error messages from the screen to an error file.

For Example: You could enter the IPFC command like this:

```
IPFC MYHELP.IPF > MYHELP.ERR
```

If you have also requested that a cross-reference list be created by specifying the /X parameter, it also will be included in the MYHELP.ERR file.

Table 17-1. NLS Language Files

ID	Language	NLS File
DAN	Danish	IPFDAN.NLS
DEU	German	IPFDEU.NLS
ENG	English UK	IPFENG.NLS
ENU	English US	IPFENU.NLS
ESP	Spanish	IPFESP.NLS
FIN	Finnish	IPFFIN.NLS
FRA	French	IPFFRA.NLS
FRC	Canadian French	IPFFRC.NLS
ITA	Italian	IPFITA.NLS
NLD	Dutch	IPFNLD.NLS
NOR	Norwegian	IPFNOR.NLS
PTG	Portuguese	IPFPTG.NLS
SVE	Swedish	IPFSVE.NLS
UND	User defined	IPFUND.NLS

Table 17-2. NLS Country Codes and Code Pages

Country	Country Code	Code Pages Supported
Australia	061	437,850
Belgium	032	437,850
Canadian English	001	437,850
Canadian French	002	863,850
Denmark	045	865,850
Finland	358	437,850
France	033	437,850
Germany	049	437,850
Italy	039	437,850
Latin America	003	437,850
Netherlands	031	437,850
Norway	047	865,850
Portugal	351	860,850
Spain	034	437,850
Sweden	046	437,850
Switzerland	041	437,850
United Kingdom	044	437,850
United States	001	437,850

Location of .HLP Libraries

When users first install OS/2 Version 1.2, the following environment variable is placed in their CONFIG.SYS files:

HELP=C:\OS2\HELP

The HELP environment variable identifies the location of .HLP libraries.

A Reminder

Common User Access (CUA) guidelines recommend that the help information described below be included as part of an application's help interface.

Keys Help

The **Keys help** selection from the **Help** pull-down should display a panel that describes key combinations used for entering application commands. For an example of a keys help panel, see the HELP1.IPF source file in the HELP1 sample program directory.

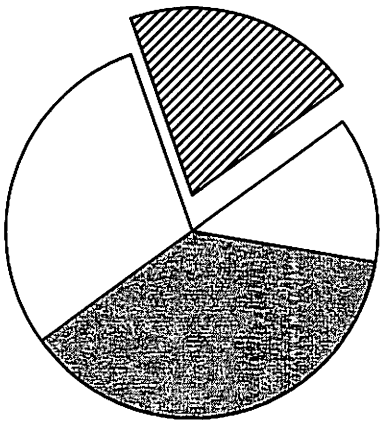
Help for Help

The **Help for help** selection in the **Help** pull-down can display a panel that describes the types of help provided by IPF and how they are accessed by the user. This panel is shipped with IPF. You may decide to replace this panel with information that is specific to the help interface of your application.

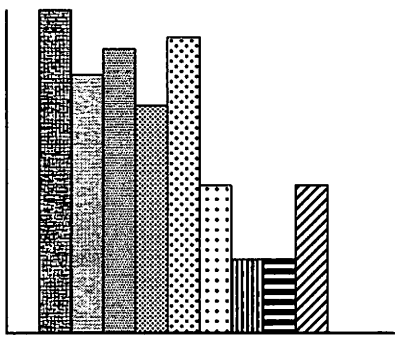
Part 4. Presentation Manager Applications – Using the Graphics Programming Interface

Chapter 18. An Introduction to the Graphics Programming Interface

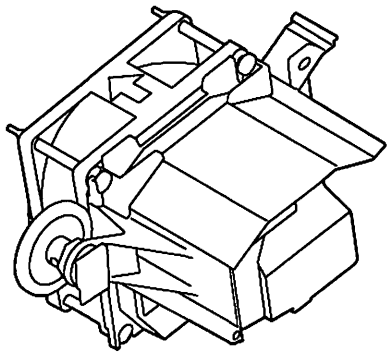
You are probably familiar by now with the visual parts of the Presentation Manager's user interface. The programming functions that the Presentation Manager uses to draw windows, icons, dialog boxes and scroll bars, for example, are also available to you, so that you can produce windows-based *graphics* applications. The Presentation Manager graphics programming interface (GPI) is an extensive one, and can be used in a variety of applications. You can use the GPI to present numerical data, such as the contents of a spreadsheet, in a way that is instantly accessible and more easily digested than rows of figures. For example, statistics can be presented as pie charts:



or as bar charts:



In addition to presenting figures in a visually striking way, you can use the GPI to produce engineering diagrams, such as:



You can even use it to produce:

Graphics Text.

Most of the graphics calls are mentioned in this part of the book. They all begin with the letters 'Gpi' and you can find them listed alphabetically in the *Programming Reference: Volume 1*. Most of the calls conform to SAA (Systems Application Architecture).

The Presentation Manager graphics interface is a *modal* interface. In nonmodal graphics systems, you supply information relevant to an instruction when you issue that instruction. For example, you could issue a line-drawing instruction on which you would specify that the line is to be red, dotted, and drawn from point a to point b. Those values or *attributes* are bound to a particular drawing instruction, and affect no other.

In modal systems, information that modifies the instruction (such as the line type and its color) is established before you issue the instruction. A line is drawn in the *current color*, in the *current line type*, and starting from the *current position*. The only value you supply on input to the line-drawing call is the end point of the line. There are separate calls for specifying the current color, line type, and position.

Current graphics values all have default settings. For example, lines are solid by default. In specific circumstances, some of the GPI calls cause current values, such as current line type, to be reset to their default values. Appendix C, "Graphics Attributes" provides more information about the current graphics values, and describes the circumstances in which they are reset to their default values.

It is possible to define graphics, and store their definitions, without sending them to a screen or a printer. The concept of *currency*, however, is relevant only when you are drawing graphics to an output device. That is, the color that is current when you draw a line on the screen is the color in which the line is drawn, *not* the color that was current when you defined the line.

The GPI Query Calls

The Presentation Manager provides GPI calls for retrieving information about current graphics values. All of these calls begin with the characters 'GpiQuery'. For example, the GpiQueryColor (GSQCOL) call returns the color that is currently being used for drawing. Not all GpiQuery calls have been mentioned in this book. However, as a general rule, for every call that you use to specify a current value, such as color, there is an equivalent query call. You can find them all in the *Programming Reference: Volume 1.*

The Sample Programs

Among the sample programs in the OS/2 Programming Tools and Information are some that illustrate commonly used graphics-programming techniques. Extracts from those sample programs are included at strategic points in this part of the book. You are recommended to print the source listings and use them for general instruction. The programs that illustrate graphics programming are:

- Graphic1
- Graphic2
- Fonttest
- Bmap
- Image.

Program Structure

Graphics output can be slow, so you should perform lengthy drawing operations on an asynchronous thread. If you do this, the main thread remains free to monitor its message queue and stay responsive to user input. Because user input goes only to the main thread, good communication between the main thread and the drawing thread is extremely important if a process is to respond rapidly to user requests. For example, if a user changes or deletes graphics, the main thread must be able to interrupt the processing of the asynchronous thread and pass on the revised instructions. The main thread does this by issuing the GpiSetStopDraw (GSSDW) call with the value SDW_ON, which suspends drawing on all threads to the graphics presentation space named on GpiSetStopDraw, and establishes the *stop-draw condition.* (Drawing to other presentation spaces in the same process and in different processes is unaffected by this stop-draw condition.) No drawing can start from another thread to the named presentation space while the stop-draw condition applies. Any drawing operations that are in progress on other threads of the same process to the presentation space, when you issue this call, are terminated with a warning. When the main thread receives an acknowledgment from the drawing thread, it can clear the suspension by issuing GpiSetStopDraw (GSSDW) with the value SDW_OFF. This technique is illustrated in the sample programs Graphic1 and Graphic2.

Chapter 19. The Graphics Presentation Space

The graphics presentation space is a user-defined area in which graphics are created before they are sent to an output device, such as a printer or a screen, and in which other information, relevant to those graphics, is stored. Because most of the GPI calls operate on a graphics presentation space, defining it is always your first task in a graphics application.

There are three types of graphics presentation space: *normal*, *micro*, and *cached micro*.

When you define a normal presentation space, all the GPI functions are available to you. For example, you can *retain* graphics, so that they can be drawn more than once during a single execution of an application without having to be recreated. A disadvantage of normal presentation spaces is that, if they are allowed to accumulate, they can use up storage and eventually degrade the performance of your system.

When you define a micro presentation space, only a subset of the GPI functions is available to you. You cannot retain graphics in a micro presentation space, for example. The benefits of a micro presentation space are reduced storage use and, because it takes less time to allocate than a normal presentation space, improved performance. A micro presentation space is generally used for simple drawings directed, once only, to a single output device. Those GPI calls that you cannot use in a micro presentation space are identified in Appendix B, "GPI Calls." A micro presentation space, like a normal presentation space, can be used for drawing on any type of output device.

Presentation Manager keeps a limited number of micro presentation spaces in an area of storage from where they can be quickly allocated. These are known as cached micro presentation spaces, and they can be used only when you are drawing to a screen window. Cached micro presentation spaces offer the best performance, because, unlike normal and micro presentation spaces, they are not permanently allocated to an application. A presentation space is required for each output device (including each Presentation Manager window) currently in use by an application. If your application uses a large number of windows, a presentation space allocated permanently to each of them would quickly deplete Presentation Manager's resources. Therefore, unless the function you require is available only in a normal presentation space, you should apply the following order of preference when defining a presentation space:

1. Cached micro presentation space (for windows only)
2. Micro presentation space
3. Normal presentation space.

The remainder of this chapter describes how to define a normal or noncached micro presentation space, how to transfer graphics from the presentation space to the output device, and how to position drawings using graphics coordinates. Cached micro presentation spaces are described on page 19-6. The chapter finishes with some example code that shows how to create a normal graphics presentation space.

Defining the Presentation Space

To define a normal or noncached micro presentation space, you use the GpiCreatePS (GIPCRT) call. As input to GpiCreatePS, you supply the height and the width of the area in which you want to draw, which is known as the *presentation page*. The presentation page is an internal representation of the paper or the screen area to which you are directing output. It is described on page 19-3. The output from the GpiCreatePS (GIPCRT) call is the *graphics presentation-space handle*, which uniquely identifies the presentation space and which must be provided as an input parameter on all GPI operations on that presentation space. Any presentation space created using GpiCreatePS (GIPCRT) is permanently allocated to the application.

A single application can create any number of presentation spaces using GpiCreatePS (GIPCRT). Each presentation space can be used only on the process on which it was created and, with the exception of the GpiSetStopDraw (GSSDW) call, can be accessed by only one thread at a time. Therefore, GPI calls can be made from only one thread, or access to the presentation space has to be serialized using semaphores. Failure to do this causes the PMERR_PS_BUSY error condition to be raised.

The Device Context

The graphics presentation space contains all the information required to produce a drawing that is device-independent. Apart from the detail of the drawing itself, this includes resources, such as color definitions, that affect the appearance of the drawing. Because the requirements of each of the possible output devices are different, they are addressed separately. All information that is device-specific is held apart from the presentation space in a *device context*. The device context is a logical description of the device, and includes the device-driver code for running the device.

Each output device you intend to use must be described in a device context. Output devices include not only the usual printers and displays, but also windows, which are a subdivision of the screen device. If a single application uses more than one window, each window has its own device context. A single physical device generally can be used by more than one application. The same device can therefore be described in any number of device contexts.

Like presentation spaces, device contexts are user-defined, although you do not use GPI calls to define them. Device contexts for windows are obtained using the WinOpenWindowDC (WIOPDC) call, and device contexts for all other output devices are created using the DevOpenDC (WDOPEN) call. A device context is identified by a *device-context handle*.

Associating a Presentation Space with a Device Context

A presentation space must be *associated* with a device context before a drawing can be printed or displayed. Before you can draw a picture in a Presentation Manager window, for example, a graphics presentation space must be associated with the device context of the window. To do this, you use the GpiAssociate (GSASS) call. On input to GpiAssociate, you supply both the graphics presentation-space handle and the device-context handle. When a presentation space and a device context have been associated, any output you direct to the presentation space is automatically sent to the device context, and from there to the physical device.

You can associate a graphics presentation space with a device context at the time the presentation space is created. You do this by supplying the device context handle as an input parameter to the GpiCreatePS (GIPCRT) call, and by setting the GPIA_ASSOC bit of the PS_ASSOCIATE option. This is the only way of associating a noncached micro presentation space with a device context. A micro presentation space cannot be reassociated.

A single graphics presentation space can be associated with only one device context at a time, and a single device context can be associated with only one graphics presentation space at a time. Before you can associate a presentation space with a different device context, you have to disassociate it from the current device context using the GpiAssociate (GSASS) call with a null device context handle. A micro presentation space cannot be disassociated. If you display a drawing in a window and then decide to print it, for example, you disassociate the presentation space from the window device context, and associate it with the printer device context. This sequence is shown in Figure 19-1.

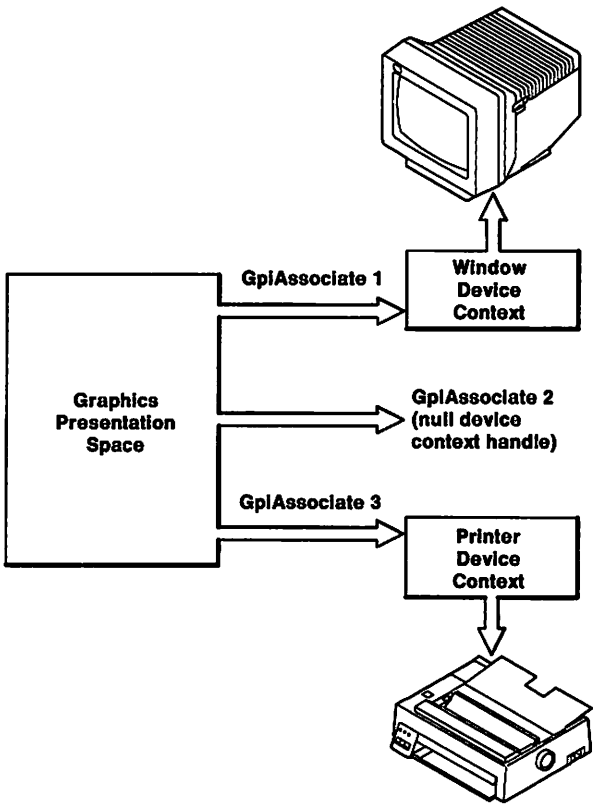


Figure 19-1. Association

In Figure 19-1 on page 19-2, the presentation space is associated first with a window device context. It is then disassociated from the window device context and associated with a printer device context. It cannot be associated with both device contexts simultaneously.

You should delete any permanently allocated presentation space that is no longer required. Before you can delete a normal presentation space, you must disassociate it from the current device context. To delete a normal or noncached micro presentation space, you use the GpiDestroyPS (GIPDEL) call, which deletes the presentation space and releases any resources associated with it.

When you have finished using a device context, you should close it using the DevCloseDC (WDCLS) call. The device context must not be associated with a presentation space when you close it, and DevCloseDC must not be used to close a window device context. A window device context is closed automatically when its associated window is destroyed.

The Presentation Page

The presentation page is an internal representation of the paper or of the area of the screen to which you are directing output. The GPI calls that you use to draw in the presentation page all require some positional information on input, so that your graphics can be located correctly within the presentation page and in relation to each other. This positional information is supplied in the form of (x,y) coordinates, as shown in Figure 19-2.

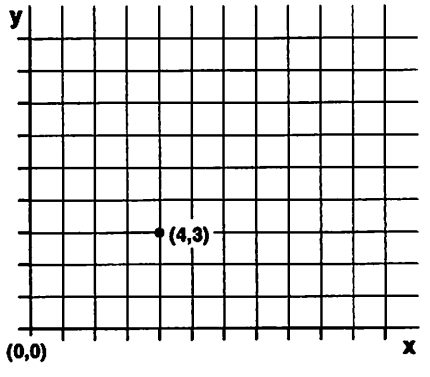


Figure 19-2. Coordinate Positions

The x axis is always horizontal, and the y axis is always vertical. The origin of a presentation space is aligned with the origin of a window (lower-left corner). Any position in the presentation space can be described as the intersection of notional lines drawn at right angles to the axes. The x axis position is always provided first. In Figure 19-2, position (4,3) is at the intersection of a notional line drawn from the fourth increment along the x axis with a line drawn from the third increment along the y axis.

The coordinate values that you specify when you issue a GPI drawing request are known as *world coordinates*, because they are values that are convenient to the application and that can have some relationship with real-world measurements.

The Current Position

The *current position* is the world-coordinate position at which the next drawing request begins. When you create or obtain a presentation space, the current position is set to the origin (0,0). The current position is also reset to (0,0) whenever a presentation space is associated with a different device context.

Most drawing requests update the current position. If you draw a line from the origin to (3,5), for example, the current position is updated to be (3,5). The current position can also be updated explicitly by the GpiSetCurrentPosition (GICP) call. If you do not want the first drawing in a presentation space to start at (0,0), you can issue a GpiSetCurrentPosition (GICP) request first.

Defining the Presentation Page

When you define a presentation page on the GpiCreatePS (GIPCRT) call, you specify both the sizes of the x and y axes, and the units of increment along the axes.

The Presentation Page Units

The presentation-page units of increment can be any of the following:

Pels

Choose pels (PU_PELS) if you want to work in terms natural to the screen or window. Pels are the most device-dependent units. Display devices have different resolutions and sizes, so a picture designed for one type of display may not look the same on a different type of

display. In particular, a picture's *aspect ratio* is not guaranteed to be preserved when you work in pels. The aspect ratio of a picture is the ratio of its width to its height. If the pels on a particular device are not square, the picture's aspect ratio is lost. For example, if you draw a square of 40 × 40 pels on a device that does not have square pels, the displayed picture will be a rectangle. The same problem arises when drawing circles, which are drawn as ellipses on such devices. You can find out the horizontal and vertical resolutions of a device using the DevQueryCaps (WDQCAP) call with the values CAPS_HORIZONTAL_RESOLUTION and CAPS_VERTICAL_RESOLUTION. Both values are returned to you in pels per meter.

Metrics

Metric units can be any of the following:

Units of 0.1 mm (PU_LOMETRIC)
Units of 0.01 mm (PU_HIMETRIC)
Units of 0.01 in. (PU_LOENGLISH)
Units of 0.001 in. (PU_HIENGLISH)
Units of 1/1440 in. (PU_TWIPS).

You are most likely to choose one of the metric values when you are drawing with a particular paper size in mind. This is often the case when you are producing architectural or engineering drawings, for example, or when you are producing graphics text in a specific point size.

Drawings that are defined in metric units are guaranteed to be produced in the specified size when sent to a printer or plotter. This cannot be guaranteed, however, when the drawings are sent to a screen device. Although you can attach screens of different sizes and resolutions to your PS/2, the VGA device driver cannot distinguish between them. Because of this, the number of pels in a meter is always assumed to be the same, regardless of the screen size. Therefore, on some screens a 10-mm line is 10 mm long, and on others it is not, although it is 10 mm long on all printers and plotters.

Arbitrary units

Arbitrary units (PU_ARBITRARY) are just numbers with no measurement scheme attached. For example, you can define a presentation page of 500 units along the x axis and 300 units along the y axis, without specifying what those units are. You are likely to choose arbitrary units when you want to produce completely device-independent graphics. If you are drawing a map of a large area, for example, you might want each unit to represent an increment of one kilometer or mile. Arbitrary units allow a picture to be displayed as large as possible on any device while preserving its aspect ratio.

The Presentation Page Size

The size of the presentation page is the number of those incremental units that you want to use along each axis. The available coordinate range depends on the coordinate format that you specify on GpiCreatePS (GIPCRT). If you select a format of GPIF_SHORT, you can use world coordinates in the range −32 768 through +32 767. If you select a format of GPIF_LONG, which is the default value, you can use world coordinates in the range −134 217 728 through +134 217 727. The format consequently has an effect on the amount of storage occupied by a presentation space. The size of your presentation page can be anywhere in the appropriate range. One approach is to find out the size of the output area of the current device, and define a presentation page of the same size. That is, if you are drawing in a screen window, you issue the WinQuerySysValue (WMQSYS) call to establish the size of the client area of the desktop window, and define a presentation page of the same size. You can use the DevQueryHardcopyCaps (WDQHC) call to establish an equivalent value for hard-copy devices.

Alternatively, you can ignore the dimensions of the device and define a presentation page of any size within the appropriate coordinate range. What can be displayed or printed without first manipulating the drawing, however, is still determined by the range available to the currently associated device. You could define a presentation page of pels, for example, but specify a size of 1500 by 1200. Your PS/2 monitor might be 640 pels (x axis) by 480 pels (y axis), so drawings within the range (0,0) through (639,479) can be displayed, and anything outside that range cannot, unless you apply a *transformation* to the drawing. Typically, a transformation would be applied to scale the presentation-page contents so that they fit within the available area of the device. Alternatively, a transformation can be applied to move (or scroll) graphics into the coordinate range of the device. Transformation is described in Chapter 25, "Transformation."

Figure 19-3 shows the relationship between the presentation page and the output device.

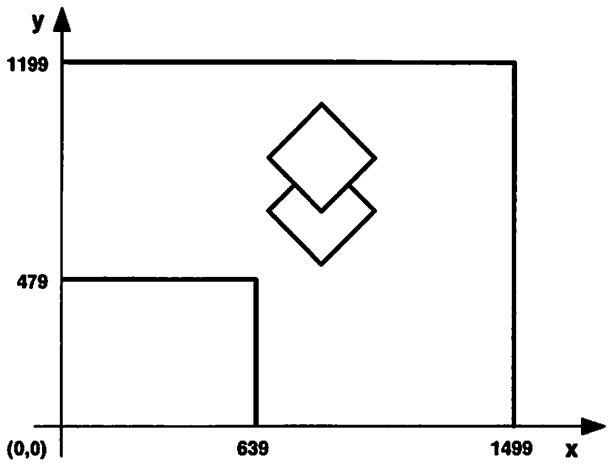


Figure 19-3. The Coordinate Range. The overlapping diamond shapes in this example are drawn within the range (0,0) through (1499,1199), which is the range of this presentation page, but they are not within the range (0,0) through (639,479), which is the coordinate range available on a particular screen device, so they have to be transformed before they can be displayed.

Presentation Manager does not prevent you from using coordinates anywhere in the range appropriate to the presentation-page format (either −32 768 through +32 767, or −134 217 728 through +134 217 727), even if you draw *outside* the presentation page whose size you specify on the GpiCreatePS (GIPCRT) call.

You can allow the presentation-page size to default. If you do this, you must associate the presentation space with a device context by providing the device-context handle on input to GpiCreatePS (GIPCRT), and setting the GPIA_ASSOC flag. Presentation Manager then defines a presentation page of the maximum size (for example, maximized window size or paper size) for that device.

In all cases, the origin of the presentation page (0,0) is at the lower-left corner of the presentation page. The origin of the presentation page maps to the origin of the associated Presentation Manager window or, if the output device is a printer, to the lower-left corner of the output area of the paper.

Mapping the Presentation Page to the Device

The area of the device in which a picture is to be displayed is known as the *page viewport*. The process by which the picture is mapped to the page viewport is called the *device transformation*, and it happens automatically.

Note: Drawing is not clipped to the page viewport or to the presentation page.

When you associate a presentation space with a device context, a default device transformation is set. By default, you are assumed to be sending output to the screen, and a page viewport of pels is defined according to the following rules:

- When the presentation page is specified in **pels**, the page viewport is the same size as the presentation page, with the lower-left corner of the presentation page mapping to the lower-left corner of the device space. For example, if you define a presentation page of 300 coordinates (x axis) by 200 coordinates (y axis), then your picture is transformed to a screen area of the same size.
- When the presentation page is specified in a **metric** unit, the page viewport is of the correct physical size (depending on the physical spacing of the pels) with the lower-left corner of the presentation page mapping to the lower-left corner of the device space.
- When the presentation page is specified in **arbitrary** units, the default size for the device is used. For a plotter or printer, this is the maximum accessible area of the paper, and for a screen, it is the maximized window size. The page viewport is constructed such that the presentation-page coordinates give equal x and y spacing. The lower-left corner of the presentation page maps to the lower-left corner of the device space, and either the right or the top edges map, such that the picture is contained within the device rectangle and its aspect ratio is preserved.

Figure 19-4 shows mapping from the presentation page to the device.

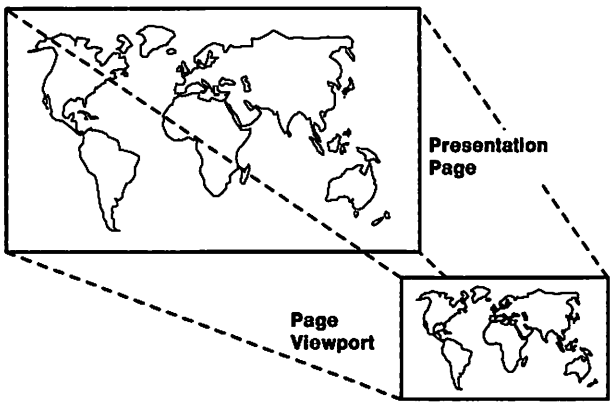


Figure 19-4. Mapping a Picture from the Presentation Page to the Device. In this example, a map of the world has been drawn in a presentation page, defined in arbitrary units, that is much larger than the device space. The device transformation scales the picture to fit the maximized window size and preserves its aspect ratio.

Note: You can also specify the device transformation explicitly using the GpiSetPageViewport (GIPV) call. This is described in Chapter 25, "Transformation."

Cached Micro Presentation Spaces

Cached micro presentation spaces are provided by the windowing system rather than by the graphics programming interface, and their use is synchronized with other windowing activities. You get a cached micro presentation space using either the WinGetPS (WPGPS) call or the WinBeginPaint (WIBPNT) call.[1] In general, you use a cached micro presentation space to process a single message. If you get the presentation space using WinGetPS (WPGPS), it can be used to process any message, but it must be returned to the cache using the WinReleasePS (WPREPS) call when the message processing is complete. That is, the presentation space must be both obtained and released during the processing of a single message. If the presentation space is obtained using the WinBeginPaint (WIBPNT) call (in response to the WM_PAINT message), it is released automatically when WinEndPaint (WIEPNT) is issued.

You have to provide a window handle on input to WinBeginPaint and WinGetPS. The resulting presentation space is defined specifically for that window, and cannot be reassociated. The presentation page of a cached micro presentation space is defined automatically by Presentation Manager. It is always defined in pels (although you can change the units using the GpiSetPS (GIPS) call), its format is always GPIF_LONG, and its size is directly related to the update region (if you are using WinBeginPaint) or to the visible region (if you are using WinGetPS or WinGetScreenPS) of the associated window. You do not have to issue the GpiAssociate (GSASS) call, because a cached micro presentation space is automatically associated with a Presentation Manager window device-context when it is obtained.

When you finish using a cached micro presentation space, you do not have to disassociate it from the window device context because the WinReleasePS (WPREPS) or WinEndPaint (WIEPNT) call performs the disassociation. This makes the cached micro presentation space available for use in other windows. The presentation space itself cannot be deleted. Cached micro presentation spaces belong to the system rather than to the application, and are allocated for use on a temporary basis. Cached micro presentation spaces are used serially: when you next require a cached presentation space, you obtain a new one using the appropriate call. Each time you get a cached micro presentation space, graphics attributes are reset to their default values.

[1] You can also get a cached micro presentation space using the WinGetScreenPS (WPGSPS) call. For information about this call, see the *Programming Reference: Volume 1*.

Example Code: Creating and Associating a Presentation Space

All the graphics sample programs show how to define a presentation space and associate it with a device context. This example is taken from the Graphic1 sample program.

```
MRESULT WndProcCreate(hwnd)
HWND  hwnd;
{
  SIZEL sizlClient;
  SIZEL sizlPickApp;

  /*********************************************************************/
  /* Establish the size of the maximum client area.                    */
  /*********************************************************************/

  sizlClient.cx = WinQuerySysValue(HWND_DESKTOP, SV_CXFULLSCREEN);
  sizlClient.cy = WinQuerySysValue(HWND_DESKTOP, SV_CYFULLSCREEN);

  /*********************************************************************/
  /* Obtain a window device context, and pass its handle as            */
  /* input to GpiCreatePS. The presentation page is the same size in   */
  /* pels as the maximum client area.                                  */
  /*********************************************************************/

  hdcClient = WinOpenWindowDC(hwnd);
  hpsClient = GpiCreatePS(habMain,
                          hdcClient,
                          &sizlClient,
                          GPIA_ASSOC | PU_PELS
                         );

  /*********************************************************************/
  /* Define a second presentation space for use on the main thread.    */
  /*********************************************************************/

  hpsPaint = GpiCreatePS(habMain,
                         (HDC)NULL,
                         &sizlClient,
                          PU_PELS
                         );

  if (hpsClient == (HPS)NULL){
      return(MRESULT)(TRUE);
  } /* endif */
```

Chapter 20. Graphic Primitives and Their Attributes

Like all graphics systems, the Presentation Manager provides a set of building blocks that you use to construct pictures. These building blocks are called *graphic primitives*. The primitives that the Presentation Manager provides are:

- The *line* primitive
- The *marker* primitive
- The *image* primitive
- The *area* primitive
- The *arc* primitive
- The *character string* primitive.

Each of these graphic primitives has a set of properties that further define its appearance. A line, for example, can be drawn in different colors, in different widths, and can be dotted, solid, or even invisible. These properties are known as *primitive attributes*. In general, each of the primitives has its own attributes, and these are set for the primitive type rather than for a single instance of the primitive. If you set the line-type attribute to LINETYPE_DOT, for example, all lines drawn subsequently are drawn as dotted lines. When an attribute value has been set it becomes the *current attribute value*, and remains in effect in the presentation space until it is next changed.

For all primitive attributes there are default settings that apply when a presentation space is first created, and that continue to apply until you change them. Attribute settings also revert automatically to their default values at other specific times. For example, whenever you issue the GpiAssociate (GSASS) call to associate a presentation space with a device context, primitive attributes revert to their default values. The initial default settings are specified by the Presentation Manager. However, you can change the default settings by using the GpiSetDefAttrs (GIDATR) call. For more information about the circumstances in which attributes are reset to their default values, see Appendix C, "Graphics Attributes."

Not all graphic primitives and attributes are described in this chapter. The color attribute and the mix attribute apply to all of the graphic primitives, and are described separately in Chapter 21, "The Color and Mix Attributes." The character-string primitive and its attributes are described in Chapter 23, "Graphics Text."

This chapter describes the line, marker, image, area, and arc primitives, and introduces the GPI calls that you use to draw them. It also describes the attributes of these primitives (excluding the color and mix attributes) and their default settings, and discusses the ways in which attribute values can be set.

The Line Primitive

There are three variants of the basic line primitive. These are the line, the polyline, and the box.

The GpiLine (GILINE) call draws a line from the current position to a specified end point. When you have drawn the line, the current position is then at the specified end point, as shown in Figure 20-1.

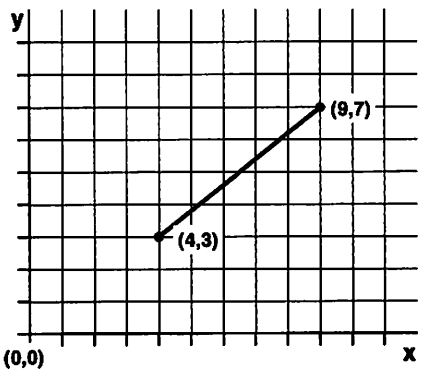


Figure 20-1. The Line. The current position is (4,3), and the specified end point of the line is (9,7). When the line has been drawn, (9,7) becomes the current position.

To draw a single point, you can issue a GpiLine (GILINE) call on which the current position and the end point of the line are the same. The GpiPolyLine (GIPLNE) call draws a sequence of one or more connected lines, starting at the current position and passing through a series of specified coordinate positions. For example, if you are drawing a line graph that has five connected lines, you can issue one GpiPolyLine (GIPLNE) call rather than five separate GpiLine (GILINE) calls. When you have issued GpiPolyLine (GIPLNE), the current position is at the end point of the last line in the sequence, as shown in Figure 20-2.

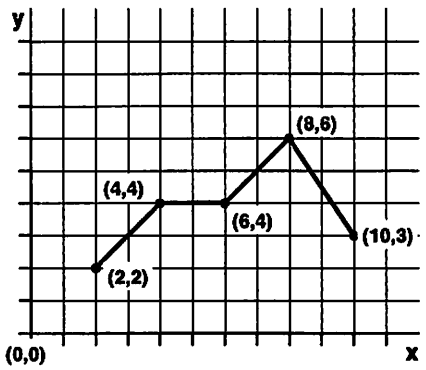


Figure 20-2. The Polyline. The starting position is (2,2), and the polyline is drawn through (4,4), (6,4), (8,6), and (10,3). The new current position is (10,3).

At its simplest, the GpiBox (GIBOX) call draws a rectangular box with one corner at the current position and the diagonally-opposite corner at a position that you specify. The sides of the box are parallel to the x and y axes. Like GpiPolyLine, GpiBox (GIBOX) lets you draw a number of connected lines using a single call rather than four separate GpiLine calls. The current position is unchanged by GpiBox. The box primitive is shown in Figure 20-3.

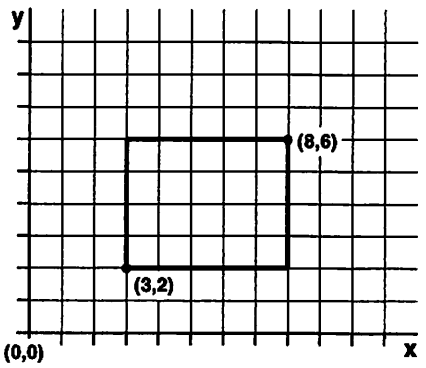


Figure 20-3. The Box. The current position is (3,2) and the second position is at (8,6). When the box has been drawn, the current position remains at (3,2). The start position and end position of any closed shape are always the same.

The box primitive can also be drawn with rounded corners. To round the corners of a box primitive, you specify, on input to GpiBox (GIBOX), the length in world coordinates of the axes of the ellipse that determines the extent to which the corners will be rounded. The ellipse can be of any size; the larger the ellipse, the greater the degree of rounding of the corners. Figure 20-4 shows how the ellipse is used to round the corners of the box.

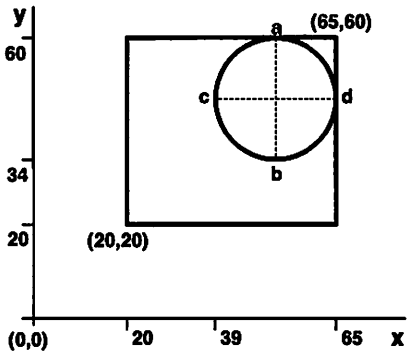


Figure 20-4. Defining a Box with Rounded Corners

The horizontal and vertical axes of the ellipse (cd and ba) have each been defined as 26 world coordinate units long. The resultant circle is tangential to the two lines that form each corner of the box and defines the degree to which the corners are rounded. Each of the four corners of the box is rounded in this way. The box with rounded corners is shown in Figure 20-5.

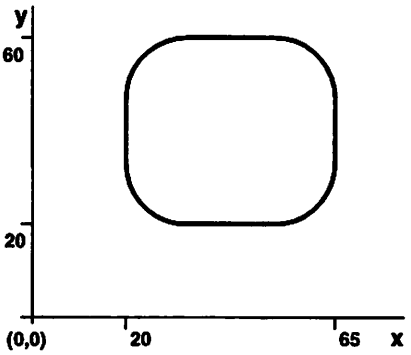


Figure 20-5. Box with Rounded Corners

Any box primitive can be drawn with a filled interior, and a filled box can be drawn without its boundary line. The box is filled with an *area-fill pattern*, such as diagonal lines or solid shading. Area-fill patterns are described on page 20-12.

Attributes of Lines

Two of the attributes of the line primitive are the *line type* and the *line width.*

To select a line type for all line primitives, you use the GpiSetLineType (GSLT) call. If you change the line type to 'dotted', for example, any line primitive drawn subsequently is drawn as a dotted line. The Presentation Manager provides nine standard line types, which are:

- LINETYPE_DOT (dotted line)
- LINETYPE_SHORTDASH (short-dashed line)
- LINETYPE_DASHDOT (dash-dot line)
- LINETYPE_DOUBLEDOT (double-dotted line)
- LINETYPE_LONGDASH (long-dashed line)
- LINETYPE_DASHDOUBLEDOT (dash-double-dot line)
- LINETYPE_SOLID (solid line)
- LINETYPE_ALTERNATE (alternate pels on)
- LINETYPE_INVISIBLE (invisible line).

The default line type is LINETYPE_SOLID. The supplied line-types are shown in the *Programming Reference: Volume 1.* You cannot define your own line types.

Line width is a device-dependent attribute, and the line width is, in all instances, the normal line width for the device. This is a fixed value, so the line width remains unchanged when you increase or decrease the size of a graphics object. If you want to alter the line width, or if you want the line width to be affected by changes of scale, you have to define a *path.* When you use a path to define wide lines, you have to specify a geometric width for the lines, and you have to specify how the ends and the joins of lines are to be drawn. For more information about paths and about the line-width, line-end, and line-join attributes, see page 20-18.

The Marker Primitive

Markers are symbols such as stars, dots, and crosses, that are commonly used to mark the peaks and troughs on a line graph. The marker primitive is shown in Figure 20-6.

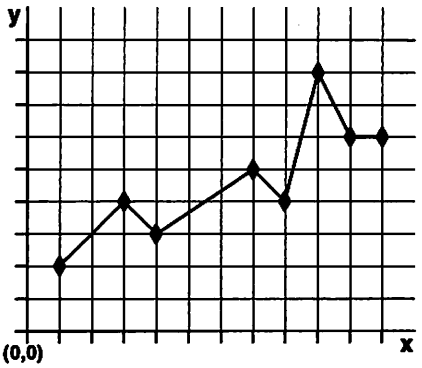


Figure 20-6. The Marker Primitive. This example shows a sequence of diamond-shaped markers drawn on a line graph at (1,2), (3,4), (4,3), (7,5), (8,4), (9,8), (10,6), and (11,6). The new current position is at (11,6).

The GpiMarker (GIMARK) call draws a single marker with its center at a specified position. This position becomes the new current position when the marker has been drawn. The GpiPolyMarker (GIMRKS) call draws a series of one or more markers, each of which is centered on a position that you specify on input to the call. All the markers drawn by a single GpiPolyMarker call are the same. When a series of markers has been drawn, the current position is at the center point of the last marker in the series.

Attributes of Markers

Two of the attributes of the marker primitive are the *marker symbol* and the *marker box.*

The GpiSetMarker (GSMS) call selects the current marker symbol from the current marker set. When the current marker has been selected, it is used on all subsequent GpiMarker and GpiPolyMarker calls until it is next changed. The Presentation Manager provides a base marker set containing:

- MARKSYM_CROSS (cross)
- MARKSYM_PLUS (plus sign)
- MARKSYM_DIAMOND (diamond)

- MARKSYM_SQUARE (square)
- MARKSYM_SIXPOINTSTAR (six-point star)
- MARKSYM_EIGHTPOINTSTAR (eight-point star)
- MARKSYM_SOLIDDIAMOND (solid diamond)
- MARKSYM_SOLIDSQUARE (solid square)
- MARKSYM_DOT (dot)
- MARKSYM_SMALLCIRCLE (small circle)
- MARKSYM_BLANK (blank).

The default marker symbol is MARKSYM_CROSS. The markers from the base marker set are shown in the *Programming Reference: Volume 1*. The supplied marker set contains image (raster) markers only. You can also create image marker symbols using the Presentation Manager-supplied Font Editor. For more information, see Chapter 23, "Graphics Text."

The marker box is a rectangular boundary that defines the horizontal and vertical space occupied by the marker symbol, and is expressed in world coordinates. To change the size of the marker box, you use the GpiSetMarkerBox (GIMB) call. The marker symbol is positioned using the marker box. That is, the center of the marker box is the center of the marker symbol. When you change the size of the marker box, a vector marker is automatically scaled up or down to fit the box. Image markers, however, cannot be scaled.

The Image Primitive

The image is a rectangular area of pels, each of which can be set on or off to produce a single-color screen image. The GpiImage (GSIMG) call defines a rectangular image with its upper-left corner at the current position. As input to this call you supply:

- The width and the depth of the image in pels.
- The setting (on or off) of each of the pels in the rectangle.

You supply the settings of the pels, starting from the top row of the image and working from left to right, with one bit describing each pel setting. The current position is not altered by this call. Figure 20-7 shows an example of the image primitive.

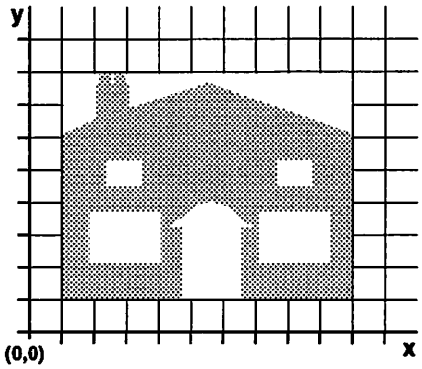


Figure 20-7. The Image Primitive. The current position of this image is at (1,8), and remains there after the image has been drawn. The image is 90 pels wide and 70 pels high.

The rectangular boundary of an image does not appear on the output device, but has been included in Figure 20-7 to show the part of the output area defined by this GpiImage (GSIMG) call. Within this rectangular boundary, specific pels are switched on to produce the image of the house. Because images are *bit maps* they are dependent on the resolution of the output device for their quality. You might find some difference in quality when you send an image that was defined for a high-resolution display to a raster printer that has a different resolution.

Attributes of Images

The image primitive has no attributes other than the color and mix attributes. These are described in Chapter 21, "The Color and Mix Attributes."

Example Code

The following example code is part of the Image Sample Program. It shows how to set up the image presentation space and memory requirements. The GpiImage call is then issued to draw the image.

```
/*************** Start of INITIMAGE Private Function *****************/
/*********************************************************************/
/* INITIMAGE:                                                        */
/* This function obtains a window device context, defines the image  */
/* presentation space, and associates the two.                       */
/* It also sets current background and foreground colors, and the    */
/* background-mix mode for the presentation space.                   */
/* INITIMAGE is invoked from WNDPROCA.                               */
/*********************************************************************/

VOID InitImage( hwndClient )
HWND hwndClient;
{
 SIZEL  sizl;

 sizl.cx = 0L;                  /* set size to default for device    */
 sizl.cy = 0L;                  /*  (full screen)                    */

 hdcImage = WinOpenWindowDC( hwndClient );
 hpsImage = GpiCreatePS( habImage
                   , hdcImage
                   , &sizl
                   , (ULONG)PU_PELS | GPIT_NORMAL | GPIA_ASSOC
                   );


 GpiSetColor( hpsImage, lForeClr );
 GpiSetBackColor( hpsImage, lBackClr );
 GpiSetBackMix( hpsImage, BM_OVERPAINT );
 }

/***************** End of INITIMAGE Private Function ******************/

/*************** Start of LOADIMAGE Private Function ******************/
/*********************************************************************/
/* LOADIMAGE:                                                        */
/* This function determines the size of the image to be loaded, and  */
/* allocates sufficient application storage to contain the image data. */
/* LOADIMAGE is invoked from the private function WNDCOMMAND.        */
/* It invokes the private functions CALCWNDMAX and INITMEM.           */
/*********************************************************************/

BOOL LoadImage()
{
  ULONG    ulBufSiz;          /* size of buffer required for the image */
  USHORT   cByte;

  struct _FILEFINDBUF InfoBuf;    /* DOS file-search results buffer    */

  HFILE      hFile;
  USHORT     usAction;
  USHORT     usAttrib  = 0;
```

```
USHORT     cSearch   = 1;
USHORT     fsOpen    = OF_OPENIFEXIST    | OF_FAILIFNEXIST;
USHORT     fsMode    = OM_NORMALFILENAME | OM_USEDOSCACHE   |
                       OM_RETERRCODE     | OM_PRIVATEHANDLE |
                       OM_EXCLUSIVEACC   | OM_READONLY;
BOOL       fFailed = FALSE;

HDIR       hdir      = 0xffff;  /* allocate a handle to caller      */
CHAR       szTmp[10]  /*to hold image file header        */


/*********************************************************************/
/* Locate the selected file and determine its size, so that enough   */
/* storage can be allocated.                                         */
/*********************************************************************/

DosFindFirst( pszFile
            , &hdir
            , usAttrib
            , &InfoBuf
            , (USHORT)( sizeof(InfoBuf) * cSearch )
            , &cSearch
            , (ULONG)NULL );


/*********************************************************************/
/* Obtain application storage to hold the image. The amount of       */
/* storage required depends on the size of the image, which is       */
/* recorded in the image-file header.                                */
/* If the image is greater than 64KB, then only the first 64K of     */
/* image-data is loaded.                                             */
/*********************************************************************/

ulBufSiz = ( InfoBuf.cbFile > 0xFFFFL ) ? 0xFFFFL : InfoBuf.cbFile;

if( fImgLoaded )                /* free any image storage buffer     */
                                /* already in use                    */
    DosFreeSeg( SELECTOROF( pbImgBuf ));

OFFSETOF( pbImgBuf ) = 0;
DosAllocSeg( (USHORT)ulBufSiz, (PSEL)&SELECTOROF( pbImgBuf), 0 );

/*********************************************************************/
/* Open the image-data file.                                         */
/*********************************************************************/

if( DosOpen( pszFile
           , &hFile
           , &usAction
           , 0L
           , usAttrib
           , fsOpen
           , fsMode
           , (ULONG)NULL ))
    {
    fBrowse = FALSE;
    fImgLoaded = FALSE;
    DosFreeSeg( SELECTOROF( pbImgBuf ));
    return(FALSE);
    }
```

```
/*********************************************************************/
/* Read the image width and height values from the image-file header.*/
/*********************************************************************/

/* Read the image width field                                        */

DosRead( hFile
       , (char far *)szTmp
       , (USHORT)4
       , &cByte );

szTmp[4] = '\0';      /* Terminate the ASCII string        */
sizlImg.cx = atol( szTmp );     /* The image width value is in bytes */

/* Check for comma in header                                         */

DosRead( hFile
       , (char far *)szTmp
       , (USHORT)1
       , &cByte );
szTmp[1] = '\0';        /* Terminate the ASCII string      */

if( strcmp(szTmp, ","))
   {
   fFailed = TRUE;
   }
else
   {

    /* Read the image height field                                   */

    DosRead( hFile
           , (char far *)szTmp
           , (USHORT)4
           , &cByte );
    szTmp[4] = '\0';        /* Terminate the ASCII string  */
    sizlImg.cy = atol( szTmp );       /* The image height is in pels */
    }

/*********************************************************************/
/* If header not correct format then fail the load                   */
/*********************************************************************/
if( fFailed ||
    sizlImg.cx == (LONG)0 ||
    sizlImg.cy == (LONG)0)
   {
   DosClose( (USHORT)hFile );
   fBrowse = FALSE;
   fImgLoaded = FALSE;
   DosFreeSeg( SELECTOROF( pbImgBuf ));
   return(FALSE);
   }

/*********************************************************************/
/* Adjust the image height (if necessary) to keep within 64K limit.  */
/*********************************************************************/

if( ulBufSiz == 0xFFFF )
    sizlImg.cy = ulBufSiz / sizlImg.cx ;
```

```
  /*********************************************************************/
  /* Convert the image-width value to pels.                            */
  /*********************************************************************/

  sizlImg.cx *= 8;

  /*********************************************************************/
  /* Transfer the image data to application storage.                   */
  /*********************************************************************/

  DosRead( hFile
         , (char far *)pbImgBuf
         , (USHORT)ulBufSiz
         , &cByte );

  DosClose( (USHORT)hFile );

  /*********************************************************************/
  /* Invoke the InitMem function to define a bit map containing the    */
  /* image.                                                            */
  /*********************************************************************/

  InitMem();
  fBrowse    = FALSE;             /* reset the Browse option.         */
  fImgLoaded = TRUE;

  /*********************************************************************/
  /* As 'view' mode is set, scrolling should be disabled. The scroll   */
  /* bars are made invisible by ensuring that they are no longer       */
  /* object windows.                                                   */
  /*********************************************************************/
  if( WinIsChild( hwndVscr, hwndFramei ))
     {
     WinSetParent( hwndVscr, HWND_OBJECT, FALSE );
     WinSetParent( hwndHscr, HWND_OBJECT, FALSE );
     WinSendMsg( hwndFramei,
                 WM_UPDATEFRAME,
                 MPFROMLONG( FCF_VERTSCROLL | FCF_HORZSCROLL ),
                 MPFROMSHORT( 0 ) );
     }
  return( TRUE );
}

/*************** End of LOADIMAGE Private Function ******************/

/**************** Start of INITMEM Private Function *****************/
/*********************************************************************/
/* INITMEM:                                                          */
/* This function creates a memory device context, defines the memory */
/* presentation space, and associates the two.                       */
/* It also creates a bit map, selects it into the memory device      */
/* context, and draws the image data into the bit map using GpiImage.*/
/* INITMEM is invoked from the private function LOADIMAGE.           */
/*********************************************************************/

VOID InitMem()
{
  POINTL ptl;
  BITMAPINFOHEADER BmapInfo;            /* bit-map information header */
```

```
if( fImgLoaded )
   /*********************************************************************/
   /* The memory presentation space is defined to accommodate a         */
   /* particular file of image data. If an image has already been       */
   /* loaded, its resources must be deleted before a new set can be     */
   /* defined.                                                          */
   /*********************************************************************/
   {
   GpiDeleteBitmap( hBmap );
   GpiAssociate( hpsMem, (HDC)NULL);
   GpiDestroyPS( hpsMem );
   DevCloseDC( hdcMem );
   }

/*********************************************************************/
/* The memory device context is defined.                             */
/*********************************************************************/

hdcMem = DevOpenDC( habImage
                  , OD_MEMORY
                  , (PSZ)"*"
                  , 8L
                  , (PDEVOPENDATA)pszDcData
                  , (HDC)NULL );

/*********************************************************************/
/* The memory presentation space is defined and associated with the  */
/* memory device context. The presentation page is the same size in  */
/* pels as the image.                                                */
/*********************************************************************/

hpsMem = GpiCreatePS(
                     habImage
                   , hdcMem
                   , &sizlImg
                   , (LONG)PU_PELS | GPIT_NORMAL | GPIA_ASSOC
                   );

/*********************************************************************/
/* Create a bit map, in a format compatible with the memory device   */
/* context.                                                          */
/*********************************************************************/

/*********************************************************************/
/* The bit map information header is defined.                        */
/*********************************************************************/

 BmapInfo.cbFix     = 12;                /* bit map header length   */
 BmapInfo.cx        = (USHORT)sizlImg.cx;
 BmapInfo.cy        = (USHORT)sizlImg.cy;
 BmapInfo.cPlanes   = 1L;                /* number of bit planes    */
 BmapInfo.cBitCount = 1L;                /* number of bits per pel  */


/*********************************************************************/
/* The bit map is defined and selected into the memory device        */
/* context. The output display is cleared.                           */
/*********************************************************************/
```

```
    hBmap = GpiCreateBitmap( hpsMem
                           , &BmapInfo
                           , 0L                          /* no options      */
                           , (PBYTE)NULL                 /* no initial data */
                           , (PBITMAPINFO)NULL
                           );


    GpiSetBitmap( hpsMem, hBmap );
    GpiErase( hpsMem );
    GpiSetBackMix( hpsMem, BM_OVERPAINT );

   /*********************************************************************/
   /* The current position is set to the top left corner of the memory  */
   /* presentation page, before GpiImage is issued to draw the image    */
   /* into it.                                                          */
   /*********************************************************************/

    ptl.x = 0;
    ptl.y = sizlImg.cy;
    GpiSetCurrentPosition( hpsMem, &ptl );
    GpiImage( hpsMem,
              0L,
              &sizlImg,                          /* image size         */
              (LONG)(sizlImg.cx * sizlImg.cy/8), /* data length in bytes*/
              pbImgBuf );                        /* image data         */

    GpiSetBitmap( hpsMem, (HBITMAP)NULL ); /* free bit map in memory DC */

    WinInvalidateRect( hwndImage, (PRECTL)NULL, FALSE );
}

/****************** End of INITMEM Private Function *******************/
```

The Area Primitive

The area primitive is any closed shape you can define, using the line and arc primitives, that is filled or shaded with a pattern, such as diagonal lines or solid color. The area primitive can contain other completely closed shapes, although it cannot contain box or full arc primitives that have themselves been defined as filled. An example of the area primitive is shown in Figure 20-8.

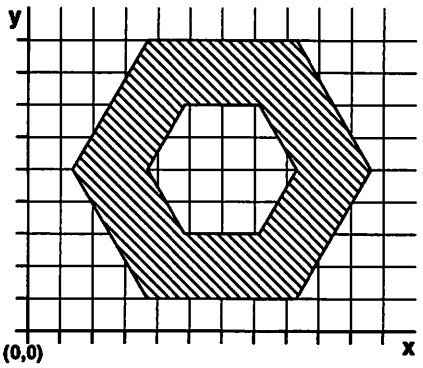


Figure 20-8. The Area Primitive. This area primitive comprises two hexagons, one completely enclosed by the other. The area is filled with the current area-fill pattern.

The GpiBeginArea (GSAREA) call signals the start of a group of primitives that define the boundary of the area. Some graphics calls are not valid within an area definition. For example, you cannot include marker, image, or character-string primitives in an area definition. Those calls that cannot be issued from an area primitive are listed in Appendix B, "GPI Calls." The current position is not changed by the GpiBeginArea (GSAREA) call, although it is updated by any drawing instructions that follow. If you issue the GpiSetCurrentPosition (GICP) call within an area definition, the current shape is automatically closed (if necessary) before the position is altered. Any figure in the area, whose start and end points are not the same, is automatically closed by drawing a straight line from the current position to the start position of the figure.

The GpiEndArea (GSENDA) call signals the end of an area definition. The current position is at the end point of the final figure in the area. The sequence of calls to draw the area in Figure 20-8 on page 20-10 is:

```
GpiBeginArea
                /* Start the area definition */
GpiSetCurrentPosition
                /* Set the start point of    */
                /* the outer hexagon         */
GpiPolyLine
                /* Draw the outer hexagon    */
GpiSetCurrentPosition
                /* Set the start point of    */
                /* the inner hexagon         */
GpiPolyLine
                /* Draw the inner hexagon    */
GpiEndArea
                /* End the area definition   */
```

An area primitive can be constructed in *alternate mode* or *winding mode*. These modes provide different ways of determining whether a given point is included in the filled area. Normally, the area-fill method you use is a matter of personal preference.

Alternate Mode

In alternate mode:

- A point is included in the filled area if you have to cross an odd number of lines to draw a line from that point to infinity.
- A point is not included in the filled area if you have to cross an even number of lines to draw a line from that point to infinity.

In the example in Figure 20-8 on page 20-10, the inner hexagon is not shaded because, to draw a line from any point in the hexagon to infinity, you must cross two boundary lines. The remainder of the area is shaded because, to move outside the area, you need to cross only one boundary line.

Winding Mode

In winding mode, the direction in which the boundary lines of the area primitive are drawn determines whether a given point is included in the filled area.

The direction of a line depends on which graphics calls you use. For example, the direction in which the GpiBox (GIBOX) call draws a box is determined by the relationship between the current position and that of the diagonally-opposite corner of the box. If the current position is at (x0,y0) and the specified corner is at (x1,y1), the box is drawn from (x0,y0) to (x1,y0) to (x1,y1) to (x0,y1) to (x0,y0). This means that in some cases the box is drawn in a counterclockwise direction (as in Figure 20-9), and in others it is drawn in a clockwise direction.

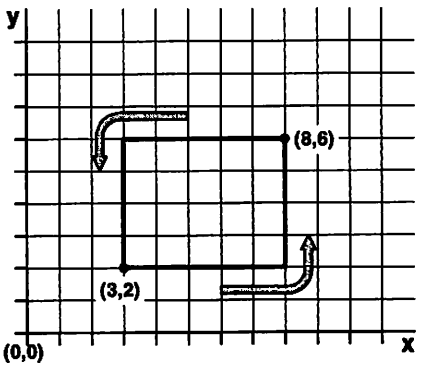


Figure 20-9. The Box. The current position is (3,2) and the specified corner is at (8,6). The box is drawn from (3,2) to (8,2) to (8,6) to (3,8) to (3,2).

For the polyline primitive, the direction in which a line is drawn depends on the relative values of the end points of the lines, so you can choose whether to draw in a clockwise or a counterclockwise direction.

To determine whether a given point is included in the filled area, you still count the number of lines to be crossed to move from that point to infinity. However, for each boundary line drawn in one direction you score plus one, and for each line drawn in the opposite direction, you score minus one. A point is within the area if the final score is nonzero. If the two hexagons in Figure 20-8 had been drawn in different directions, the final score would be zero and the area would look exactly as it does in alternate mode. If the two hexagons had been drawn in the same direction, the final score would be 2 and the inner hexagon would be shaded as shown in Figure 20-10.

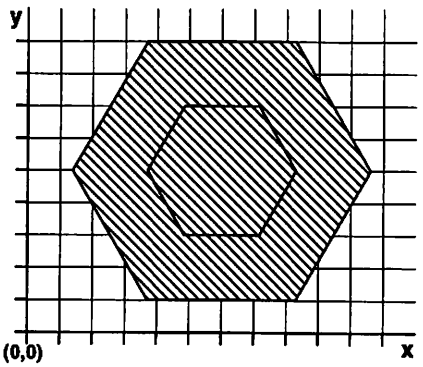


Figure 20-10. Area Constructed in Winding Mode

The boundary lines of the area in Figure 20-10 have been drawn and are visible through the area-fill pattern. The boundary lines of an area primitive do not have to be drawn. If they are, they are drawn according to the current line attributes. To vary the appearance of different parts of the boundary line, you can change the current line attributes during the course of the area definition. The boundary lines of an area are considered to be a part of the area's interior. Therefore, when you draw an area without its boundary lines (BA_NOBOUNDARY), the area-fill pattern extends to include the boundaries of the area, and the area is the same size as it would be if the boundary lines had been drawn.

Attributes of Areas

Two of the attributes of the area primitive are the *pattern symbol* and the *pattern reference point.* You use the GpiSetPattern (GSPAT) call to select an area-fill pattern from the current pattern set. The Presentation Manager provides a standard pattern set containing image (raster) patterns, which is shown in the *Programming Reference: Volume 1.*

The standard pattern set contains patterns of solid shading with decreasing intensity, and patterns of vertical, horizontal, and diagonal lines. An area is filled by replicating the pattern vertically and horizontally within the area boundary. You can also define your own image pattern sets. For information about this, see Chapter 23, "Graphics Text."

The pattern reference point is a point, expressed in world coordinates, from which the area-fill pattern spreads out horizontally and vertically. The lower-left corner of the pattern definition is aligned on the pattern reference point. The pattern reference point does not have to be within the area to be filled; indeed, the default reference point is at (0,0). Figure 20-11 shows an area filled using the default pattern reference point.

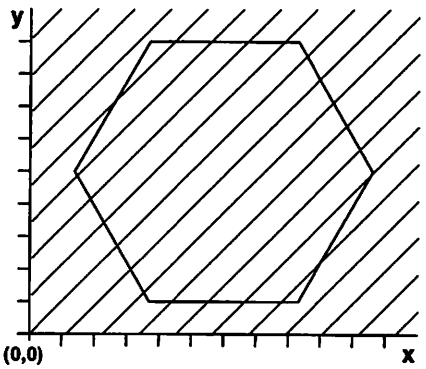


Figure 20-11. The Pattern Reference Point

The hexagon in Figure 20-11 is the area to be filled, and the pattern reference point is (0,0). This causes the diagonal lines of the pattern to intersect the area boundary at specific points. If you change the reference point to (0,4), for example, the pattern shifts upwards and the points of intersection with the area boundary are different. Although the reference point is outside the area, only the area is filled. To specify the pattern reference point you use the GpiSetPatternRefPoint (GIPRP) call.

The Arc Primitives

The arc primitive is a part, or the whole, of a circle or ellipse. There are five variants of the arc primitive, which are:

- The full arc
- The partial arc
- The 3-point arc
- The fillet
- The spline.

These different types of arc fall into two distinct groups. The second group comprises the fillet and spline, which are the most sophisticated of the arc primitives and have a different construction method from the rest. The fillet and the spline are described on page 20-16. The remaining arcs (the full arc, the partial arc, and the 3-point arc) are all defined to some extent by the *current arc parameters*.

The Current Arc Parameters

The current arc parameters define a complete circle or ellipse (the *current arc*) on which subsequent full, partial, and 3-point arcs are based. To define the current arc, you use the GpiSetArcParams (GISAP) call. GpiSetArcParams has four input values (p, q, r, and s), which define the shape, the orientation, and the size of the current arc. They also define the direction in which it is drawn.

Defining an Ellipse

To define an ellipse as the current arc, you issue a GpiSetArcParams (GISAP) request on which the values (r,q) and (p,s) are the world coordinate positions of the end points of the major and minor axes of the ellipse. For example, current arc parameters of (18,0) and (0,10) define an ellipse with a major axis of 36 coordinate units and a minor axis of 20 coordinate units. The resultant ellipse is shown in Figure 20-12.

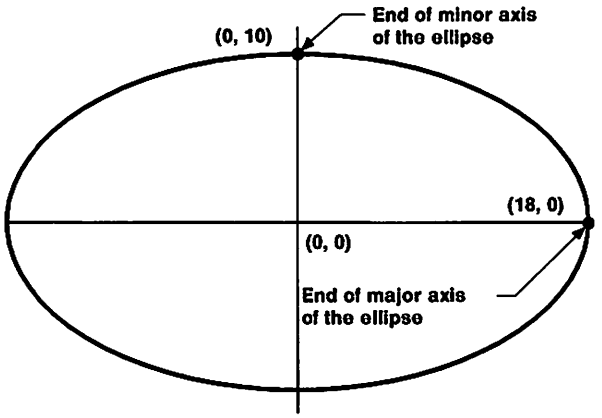


Figure 20-12. The Ellipse.

For maximum accuracy, you should aim for the axes of an ellipse to be at right-angles to each other. You can check this by ensuring that the following equation is always true:

$p.r + s.q = 0$

So, to check the above example:

$0 \times 18 + 10 \times 0 = 0$

You can also define a tilted ellipse as the current arc. None of the current arc parameters for a tilted ellipse will be zero, though you should still ensure that the axes of the ellipse are at right-angles to each other. Figure 20-13 shows a tilted ellipse defined with current arc parameters of (8,6) and (−3,4).

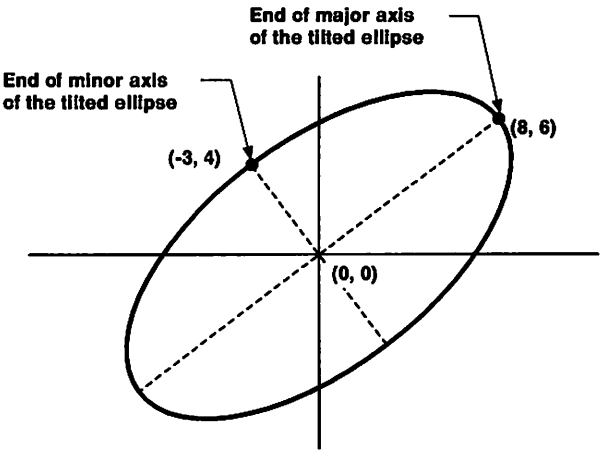


Figure 20-13. A Tilted Ellipse.

Defining a Circle

To define a circle as the current arc, the values of (r,q) and (p,s) can be any such that (r,q) and (p,s) lie on a circle. For example, if (r,q) and (p,s) are set to (−4, 3) (3,4), the current arc is a circle with a radius of 5 coordinate units. For simplicity, a circle can be defined by specifying current arc parameters where p = the radius of the circle, and $p = q$, $r = 0$, and $s = 0$. To define a circle centered on the origin and with a radius of 10 world-coordinate units, for example, you issue the GpiSetArcParams (GISAP) call with the values (0,10) (10,0).

The default values of the current arc parameters are (0,1) (1,0), which define a circle with a radius of one world-coordinate unit (a *unit circle*).

The Direction of the Arc

The current arc, and consequently any arc based on the current arc (except the 3-point arc), can be drawn clockwise or counterclockwise. The direction of the drawing is determined by the relative values of the current arc parameters:

- If $p.q > r.s$, the arc is drawn counterclockwise

- If $p.q < r.s$, the arc is drawn clockwise
- If $p.q = r.s$, a straight line is drawn rather than an arc.

Drawing a Full Arc

The GpiFullArc (GIFARC) call draws a complete circle or ellipse with its center at the current position. The current position is unchanged by this call. Whether GpiFullArc (GIFARC) draws a circle or an ellipse depends on the current arc parameters. When the current arc is a circle, GpiFullArc draws a circle, and when the current arc is an ellipse, GpiFullArc draws an ellipse. You can specify a multiplier value on GpiFullArc, so that the size of the full arc is increased or decreased in relation to the current arc. For example, if the current arc parameters define an ellipse whose major axis is 20 coordinate units and whose minor axis is 8 coordinate units, a multiplier of '2' on the GpiFullArc (GIFARC) call creates an ellipse whose major axis is 40 coordinate units and whose minor axis is 16 coordinate units.

As the default arc parameters define a circle of one world-coordinate unit, you can draw a circle of any size by allowing the arc parameters to default, and by specifying the radius of the circle on the GpiFullArc (GIFARC) call. For example, to draw a circle with a radius of 12 coordinate units, you can specify a multiplier of '12' on GpiFullArc and allow the current arc parameters to default.

You can draw a full arc that is filled with the current area-fill pattern, and you can specify whether the outline of a filled full arc is to be drawn.

Figure 20-14 shows the full arc.

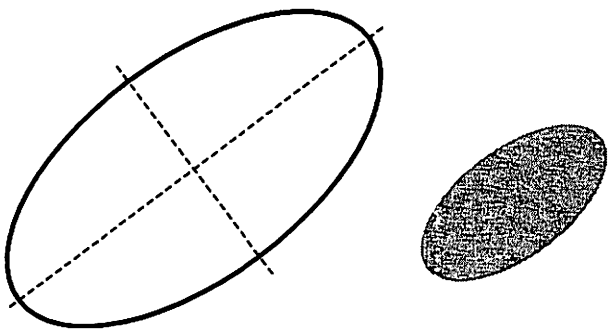


Figure 20-14. The Full Arc

Figure (a) in Figure 20-14 is a tilted ellipse that is defined by the current arc parameters. Figure (b) is drawn using the GpiFullArc call with a multiplier value that reduces the size of the arc relative to the current arc. It has also been specified as filled with the current area-fill pattern, and its outline has not been drawn.

Drawing a Partial Arc

The partial arc is a section of the full arc defined by the current arc parameters. To draw a partial arc you use the GpiPartialArc (GIPARC) call, which draws two separate figures. The first is a straight line from the current position to the starting point of a partial arc, and the second is the arc itself. When the arc has been drawn, the new current position is at the end point of the arc.

On input to GpiPartialArc (GIPARC), you specify the world coordinate position on which the current full arc (of which the partial arc is a part) is to be centered. You must also specify a start angle and a sweep angle (subtended from the center of the full arc), which determine how much of the current full arc is drawn by GpiPartialArc. You can specify a multiplier value to increase or decrease the size of the partial arc in relation to the current full arc.

Figure 20-15 shows how the partial arc is constructed.

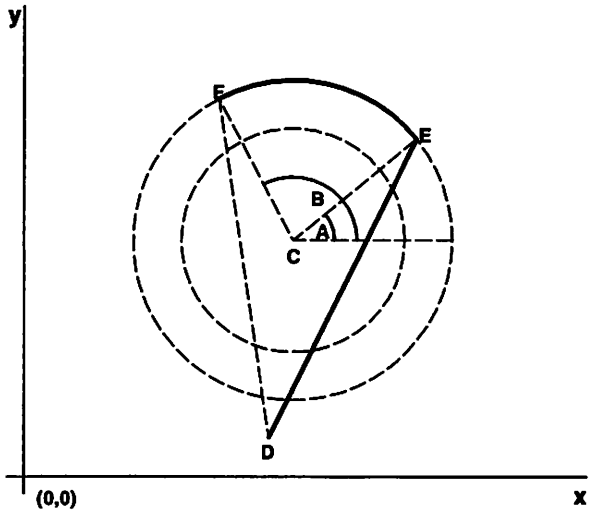


Figure 20-15. The Partial Arc

The inner circle in Figure 20-15 is the arc defined by the current arc parameters. A multiplier has been specified on GpiPartialArc (GIPARC), so the partial arc is based on the full arc described by the

outer circle. Point C is the center point specified on GpiPartialArc. Angle A is the start angle and angle B is the sweep angle. The GpiPartialArc (GIPARC) call therefore draws a line from the current position (point D) to the start of the partial arc (point E). It also draws the arc E F. Point F is the new current position. The arc is drawn counterclockwise because the current arc parameters define a counterclockwise circle. You can join points F and D using the GpiLine (GILINE) call. This line is drawn automatically if you define the partial arc within a GpiBeginArea (GSAREA) and GpiEndArea (GSENDA) bracket.

The following steps show how to use the GpiPartialArc (GIPARC) call to draw a closed shape bounded by a chord and an arc:

1. Set the current line type to LINETYPE_INVISIBLE using the GpiSetLineType (GSLT) call.
2. Issue the GpiPartialArc (GIPARC) call with the start angle = angle B, and the sweep angle = 0. This effectively moves the current position to a point on the current arc, and thereby defines one end of the chord.
3. Select a visible line type using GpiSetLineType (GSLT).
4. Issue GpiPartialArc (GIPARC), with the start angle = angle A, and the sweep angle = (angle B – angle A). Angle B should be greater than angle A. The center point is the same on both GpiPartialArc (GIPARC) calls.

To fill this partial arc with the current area-fill pattern, you can bracket the GpiPartialArc (GIPARC) call of step 4 with GpiBeginArea (GSAREA) and GpiEndArea (GSENDA) calls. You should not issue GpiBeginArea (GSAREA) before step 2.

The effect of this sequence is shown in Figure 20-16.

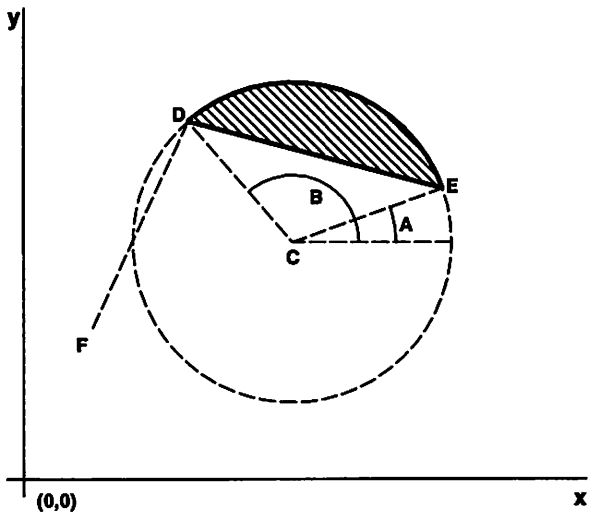


Figure 20-16. Closed Figure Bounded by Chord and Arc

The circle in Figure 20-16 is defined by the current arc parameters. Point F is the current position, and point C is the center of the arc as specified on the first GpiPartialArc (GIPARC) call. The first GpiPartialArc call (with the line type set to LINETYPE_INVISIBLE) moves the current position to point D. The second GpiPartialArc call (with the line type set to LINETYPE_SOLID) draws the chord from the current position (point D) to the start point of the arc (point E) and draws the arc E D. In this example, the partial arc has been defined within an area and has been filled with the current area-fill pattern.

Drawing a 3-Point Arc

GpiPointArc (GIGARC) draws an arc from the current position through an intermediate point to an end point. When the arc has been drawn, the current position is at the end point of the arc. You specify both the intermediate point and the end point, and these values determine both the size of the arc and the direction in which it is drawn. The shape and the orientation of a 3-point arc are determined by the current arc parameters.

Figure 20-17 shows how the 3-point arc is constructed.

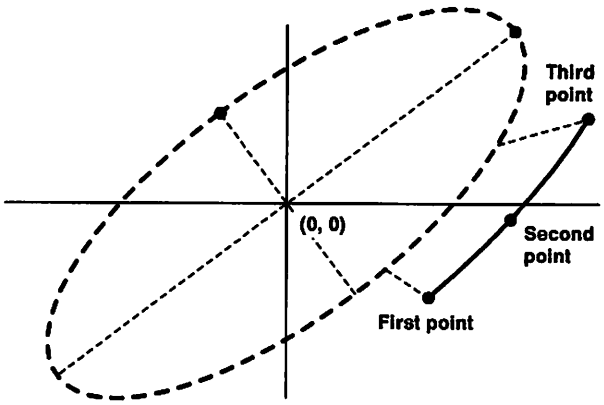


Figure 20-17. The 3-point Arc.

If the current arc parameters define an ellipse, that ellipse is scaled up or down to fit the three points of the arc.

When you want the three points of the arc to be points on a circle, allow the current arc parameters to default so they define a unit circle. You do not need to use the values p and q to specify the radius of the circle, because the radius is determined by the relative positions of the three points of the arc.

The Fillet

GpiPolyFillet (GIPFLT) constructs a fillet (a curved line) made up of one or more arcs, each of which touches a different straight line. You specify the end points of these straight lines on the GpiPolyFillet (GIPFLT) call. The lines are not drawn, but are used to construct the curve.

The fillet starts at the current position and finishes at the end point of the last line. On the way from the start point to the end point, the fillet is tangential to all intermediate lines at their *midpoints*. When the fillet has been drawn, the current position is at the end point of the last construction line. Figure 20-18 shows how the fillet is constructed.

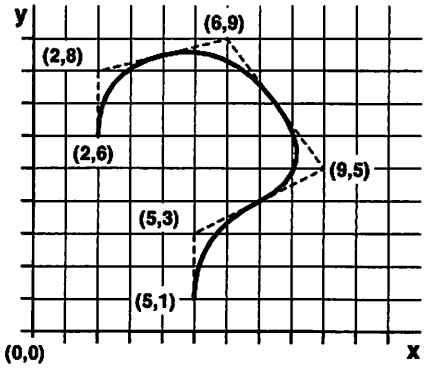


Figure 20-18. The Fillet. The fillet starts at (2,6) and ends at (5,1). The fillet is tangential to the midpoints of the lines from (2,8) to (6,9), from (6,9) to (9,5), and from (9,5) to (5,3).

When you supply only two points, the construction lines of the fillet are drawn from the current position to the first point, and from the first point to the second point. The fillet is drawn from the current position to the second point, and is tangential to the construction lines at those same points.

The GpiPolyFilletSharp (GIPFLS) call also creates a fillet on a series of connected construction lines. Each fillet, however, is built using two construction lines, one drawn from the current position to point 1 (the control point), and one drawn from point 1 to point 2 (the end point). The fillet is drawn from the current position to the end point, and is tangential to the two construction lines at those points.

You must also specify a *sharpness* value on the GpiPolyFilletSharp (GIPFLS) call. Sharpness is a measure of the distance between the fillet and the control point, and is calculated as shown in Figure 20-19.

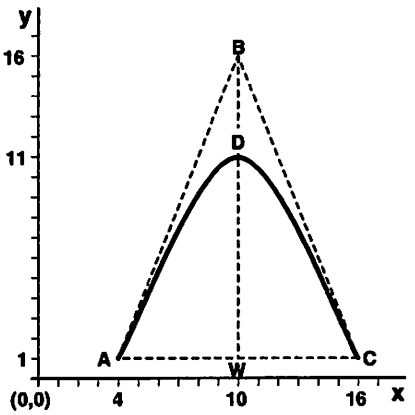


Figure 20-19. Fillet with Sharpness Specified

In Figure 20-19, point A is the current position, point B is the control point, and point C is the end point. W is the midpoint of the notional line A C. D is the point at which the fillet crosses the notional line W B. The sharpness of the fillet is the value W D / D B. The line W D is ten coordinate units, and the line D B is five coordinate units, therefore the sharpness value is 2. A sharpness value of > 1.0 defines a hyperbola, a sharpness value of 1.0 defines a parabola, and a sharpness value of < 1.0 defines an ellipse.

Subsequent fillets start from the end point of the previous fillet, and are constructed using the next two lines in the sequence in exactly the same way. Thus, for each fillet you define one control point, one end point, and one sharpness value. On completion, the current position is at the end point of the final construction line in the sequence.

There may be discontinuity of gradient between multiple fillets drawn using the GpiPolyFilletSharp (GIPFLS) call. To avoid this, points B and C of one fillet must be on the same construction line as points A and B of the next fillet in the sequence. This concept is illustrated in connection with the spline primitive in Figure 20-21. Discontinuity of gradient between fillets does not occur when the fillets are drawn using GpiPolyFillet (GIPFLT).

The Spline

The GpiPolySpline (GIPSPL) call creates a succession of one or more Bezier splines. The spline is also a curve, but its construction method is different from that of the fillet. As input to this call, you supply three construction points for each spline. The first spline starts from the current position and ends at the third specified point. The two intermediate points are control points for the curve. Subsequent splines start at the end point of the previous spline and end at the next spline point but two. Figure 20-20 shows the construction method for the spline.

To avoid discontinuity of gradient between the end of one spline and the start of the next, you must ensure that the last two construction points of the first spline and the first two construction points of the second spline are positioned along a single construction line. This concept is illustrated in Figure 20-21.

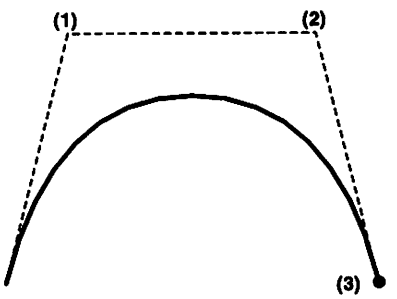


Figure 20-20. The Spline. Points (1) and (2) are the control points of the spline, and point (3) is the end point.

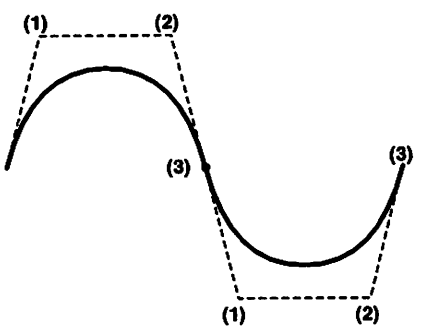


Figure 20-21. Spline with No Discontinuity of Gradient. The last two points of the first spline (points 2 and 3) are positioned along the same construction line as the first two points (current position and point 1) of the second spline.

Attributes of Arc Primitives

The arc primitive shares the line-type and line-width attributes with the line primitive. If you issue the GpiSetLineType (GSLT) call to change the line type to LINETYPE_DASHDOT, for example, all lines and arcs drawn subsequently are drawn using the dash-dot line type. Line type and line width are described on page 20-3.

Setting a Group of Attributes for a Primitive Type

Most of the attribute-setting calls discussed so far let you change a single attribute value for a single primitive type. The Presentation Manager also provides a method of setting a bundle of attribute values, including the color and mix attributes, for a single primitive type. To do this you use the GpiSetAttrs (GIATR) call, and as input to this call you provide the following:

- The primitive type
- An attributes mask
- A defaults mask
- An attributes buffer.

The *primitive type* is the primitive for which you are specifying a group of values, and can be one of the following values:

PRIM_LINE (for both line and arc primitives)
PRIM_CHAR
PRIM_MARKER
PRIM_AREA
PRIM_IMAGE.

The *attributes mask* is a 4-byte value comprising 32 flags, each of which represents a single attribute of the primitive type. For example, if the primitive type is PRIM_LINE, there is one flag to represent line type, another to represent line color, and so on. To change the value of a particular attribute, you switch on the corresponding flag in the attributes mask.

The *defaults mask* has the same format and the same flag-numbering scheme as its associated attributes mask, and its purpose is to set attributes to their default values. For example, if the line-type flag of the attributes mask is set on, and if the corresponding flag of the defaults mask is also set on, the current line-type setting is changed to its default value.

If an attribute flag is set on and its corresponding defaults flag is set off, the new setting for the attribute is taken from the *attributes buffer*. The attributes buffer has a fixed structure and is described in the *Programming Reference: Volume 1*.

To discover the current attribute values for a single primitive type, you use the GpiQueryAttrs (GIQATR) call.

Paths

The path function has a number of uses. It lets you:

- Define lines and curves with a geometric width that can be subject to changes of scale.
- Define lines and curves with a cosmetic width. This function can also be used to produce hollow characters if the object you are drawing contains characters that use an outline font.
- Define shapes for filling. This function is also available with the area primitive.
- Define irregular shapes to which subsequent primitives are *clipped*.

This section describes how to define a path, how to create wide lines, how to draw a path outline, and how to use a path as a filled shape. Clipping is described in Chapter 26, "Clipping."

Defining a Path

To signal the start of a path definition, you use the GpiBeginPath (GSPATH) call. Each path you define becomes the current path and replaces any existing path in the presentation space.

A path comprises one or more figures made up of graphic primitives.

Some GPI calls cannot be issued while there is an open path definition, and these are identified in Appendix B. In particular, you cannot include image or area primitives in a path definition. If you are using the path to define wide lines, any figure in the path whose start and end points are the same should be closed using the GpiCloseFigure (GSFCLS) call. GpiCloseFigure draws a line from the current position to the start point of the figure.

If the path definition contains full arc or box primitives, however, you must not close them using GpiCloseFigure (GSFCLS). GpiBox (GIBOX) and GpiFullArc (GIFARC) generate completely closed figures, and must not be used within another figure definition.

To signal the end of the current path definition, you use the GpiEndPath (GSENDP) call. The path definition is constructed in the currently associated device context.

Here is a typical sequence of calls to create a path containing two figures:

```
GpiBeginPath
                /* Begin the path definition */

GpiSetCurrentPosition
GpiPolyLine
                /* Draw the first figure     */
GpiCloseFigure
                /* Close the first figure    */

GpiSetCurrentPosition
GpiBox
                /* Draw the second figure    */
GpiEndPath
                /* End the path definition   */
```

Using the Path as a Filled Area

To draw the area enclosed by the current path, you issue the GpiFillPath (GSFILP) call, which draws the path and fills its interior using the current area-fill pattern. It does not draw the boundary lines of the path. Paths, like area primitives, can be filled in alternate mode or in winding mode. For a description of the area-fill modes, see page 20-11. Any open figures in the path definition are closed automatically by the GpiFillPath (GSFILP) function. When GpiFillPath is complete, the path definition is automatically deleted.

Creating Lines with a Geometric Width

In every graphics presentation space there are current geometric line-width, line-end, and line-join values. These values have no effect, however, unless they are used in the context of a path definition. By default, the geometric line-width is 1, which gives the thinnest possible line for the currently associated device. To specify a geometric line-width to be used in subsequent path definitions, you use the GpiSetLineWidthGeom (GIGLW) call. The value you specify on this call is the thickness of the line in world coordinates, and it is subject to subsequent changes of scale, or *transformations*. For example, if you specify a scaling transformation of 0.5 for an object whose current geometric line width is 4 world coordinates, the transformed line is 2 world coordinates wide. Transformation is described in Chapter 25.

Lines that have a geometric width have to be closed in some way. You use the GpiSetLineEnd (GSLE) call to specify that wide lines should have round, flat, or square ends. Line ends are flat by default. The various line ends are shown in Figure 20-22.



Figure 20-22. Closing Wide Lines

A square line-end extends the line by a distance that is half the width of the line. If the line is six coordinate units wide, a square line-end extends it by three coordinate units.

A round line-end is constructed by drawing a circle whose radius is half the width of the line.

Where one wide line joins another, the nature of the join must be defined. The GpiSetLineJoin (GSLJ) call is used to specify a beveled join, a rounded join, or a mitred join. Lines have beveled joins by default. The different line joins are shown in Figure 20-23.

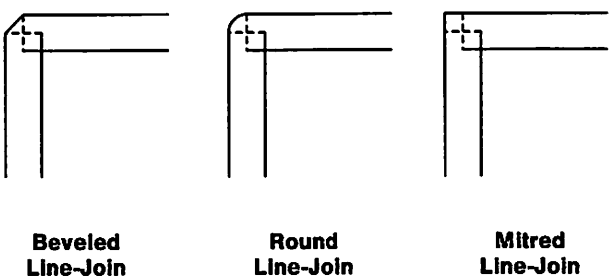


Figure 20-23. Joining Wide Lines.

To draw a figure with lines that have a geometric width, you define the figure within a GpiBeginPath (GSPATH) and GpiEndPath (GSENDP) bracket. This path becomes the current path. You then issue the GpiStrokePath (GSSTRP) call, which con-

verts the path using the current geometric line-width, line-end, and line-join values, such that the original figure is enveloped, and draws it on the current output device.

Open figures (that is, figures not explicitly closed using GpiCloseFigure (GSFCLS)) are drawn with line ends rather than line joins at their start and end points. Figure 20-24 shows the effects of the GpiStrokePath (GSSTRP) call.

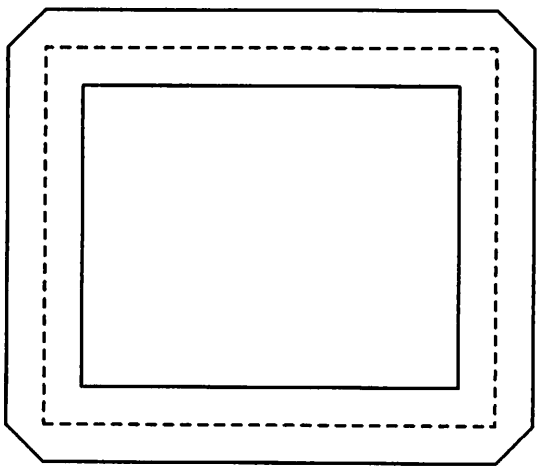


Figure 20-24. Defining Lines with a Geometric Width. The broken line is the figure defined within the path. The solid lines show the path after it has been converted. Each line has a geometric width of *n* coordinate units, and the line-joins have been defined as beveled.

GpiStrokePath (GSSTRP) automatically fills the path in winding mode with the current area-fill pattern. When GpiStrokePath is complete, the path definition is deleted from the device context.

As an alternative to the GpiStrokePath (GSSTRP) call, you can convert the path using the GpiModifyPath (GSMODP) call, which does *not* draw the path on the current output device. To draw a modified path, you issue the GpiFillPath (GSFILP) call, which draws the path and fills it with the current area-fill pattern in winding mode. You cannot fill a modified path in alternate mode.

On some devices, you may find that the GpiStrokePath method works better than GpiModifyPath followed by GpiFillPath.

Creating an Outline of a Path

If you are drawing a complex object, consisting of several character strings that use an outline font, it can take quite a while to produce. If you only want to create a rough draft of the object, you may find it quicker to draw an outline. You can do this in one of two ways:

- Using the GpiCreateLogFont call.

 This allows you to define a outline font that uses hollow characters. For more information about how to define an outline font in this way, see "Defining a Logical Font" on page 23-2.

- Using the GpiOutlinePath call.

 This allows you to define an outline path that uses hollow characters. Only use this call if you want to:

 - Clip the object
 - Shade the object with a pattern
 - Use wide lines and curves.

 Using the GpiOutlinePath call is described below.

To create the outline of a path, you use the GpiOutlinePath call. This uses the current path definition, as created with GpiBeginPath and GpiEndPath. However, it uses the cosmetic line width, rather than the geometric line width.

The GpiOutlinePath call normally has the same effect as if the lines and curves used to make up the path had been drawn without using this function. However, if the path contains character strings that use an outline font, the characters are not filled. This improves performance and may be perfectly acceptable for small characters.

Chapter 21. The Color and Mix Attributes

The color attribute and the mix attribute apply to all graphic primitives. The color attribute determines the color in which a primitive is drawn, and the mix attribute controls how each primitive is combined with the existing drawing. The mix value you choose affects, among other things, the color that results when primitives of different colors overlap.

This chapter begins with a description of the way in which colors are defined in color tables. It describes the default color table, and discusses how to replace it or change some of its contents. The remainder of the chapter describes how to specify the current color and mix attributes for graphic primitives.

Color Tables

Color display devices have color guns of red, blue, and green light that can collectively or individually illuminate the phosphors in a single pel. The pel is the smallest element of the screen that can be addressed. By switching these color guns on and off in different combinations, you can produce the following eight colors:

Black
White
Blue
Red
Pink
Green
Cyan (turquoise)
Yellow.

Yellow, for example, is produced by switching on the red and green guns, and switching off the blue gun.

Each pel is described internally by a number of bits of storage. In a monochrome display, you need only one bit per pel, and that bit is either on or off. In a simple 8-color system, you need 3 bits per pel. Each of those three bits records the on or off setting of one of the color guns. The bits are organized in *color planes*, one for each of the colors. Each color plane echoes the screen, in that there is one bit in each plane for each pel on the screen. For example, if a bit is switched on in the green and blue planes, and is switched off in the red plane, the pel that those bits describe is colored cyan.

The on and off settings required in the red, green, and blue planes to produce the eight basic colors can be summarized in the following color table:

	R	G	B
White	1	1	1
Black	0	0	0
Red	1	0	0
Blue	0	0	1
Green	0	1	0
Cyan	0	1	1
Pink	1	0	1
Yellow	1	1	0

If you want more than eight colors, you have to be able to control the *intensity* of each of the primary colors, in addition to being able to switch it off or on. To be able to vary the intensity of a color, you need more color planes for each color. For example, you could have two planes for each color rather than one, and this would give you a choice of 32 colors. The wider the range of colors you want available, the greater the demands on storage to hold the color planes of each of the colors. The solution to the storage problem is to keep the choice of colors wide, but to restrict the number of colors available to you from that choice. You define the colors that you want to use in a *logical color table*.

A logical color table contains a variable number of entries, each of which describes a different RGB (Red, Green, Blue) combination that produces a color. The principle of the color table is illustrated in Figure 21-1.

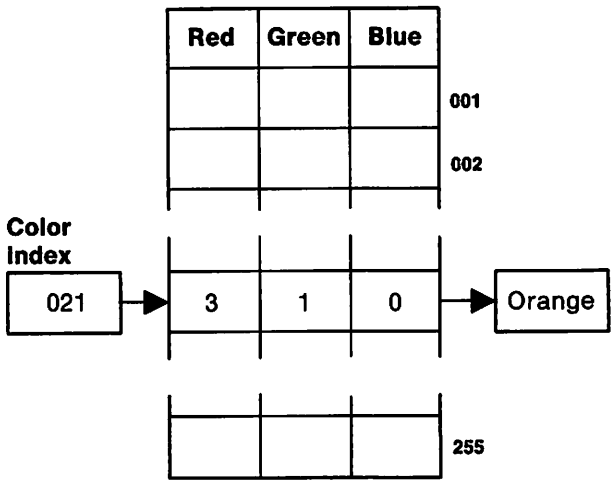


Figure 21-1. The Color Table. This simplified example conveys the message that, to produce a particular shade of orange, you need more red than green, and no blue at all. Each of the color definitions in a color table is identified by an index number. In this example, orange is addressed by index 21.

An advantage of this method of defining colors is that, as the colors you get are likely to vary from one output device to another, you can fine-tune the color definitions to get the best results on a particular output device. Any color can be made more or less intense by altering its definition in the logical color table.

Each output device has its own *physical color table*, which is organized in the same way as a logical color table. The physical color table contains the RGB color definitions of the distinct colors that the device can produce, whereas the logical color table contains the definitions of the colors you want to use. As you draw on an output device, the Presentation Manager substitutes the index value that addresses the definition of the current color in the logical color table with the index that retrieves the closest match for the current color in the physical color table.

Because this substitution takes place when the graphics are drawn, the presentation space can be associated with a number of different device contexts without invalidating the logical color definitions. For example, if you create a picture on the screen using a wide range of colors, and then direct that drawing to an 8-pen vector plotter, the drawing can be reproduced without having to be recreated. Although you do not get the variety of colors you require, the substitution process selects, from the eight colors available, the nearest match for each color you want.

The Default Logical Color Table

The Presentation Manager provides a default logical color table. The colors defined in this table, and the indexes that retrieve them, are shown in Table 21-1.

Table 21-1. The Default Color Table

Color Index	Index Number	Effect
CLR_FALSE	−5	All color planes set to 0
CLR_TRUE	−4	All color planes set to 1
CLR_DEFAULT	−3	Default value − Same as CLR_NEUTRAL
CLR_WHITE	−2	White
CLR_BLACK	−1	Black
CLR_BACKGROUND	0	Natural background color for the device
CLR_BLUE	1	Blue
CLR_RED	2	Red
CLR_PINK	3	Pink
CLR_GREEN	4	Green
CLR_CYAN	5	Cyan
CLR_YELLOW	6	Yellow
CLR_NEUTRAL	7	Neutral − Contrast color to CLR_BACKGROUND
CLR_DARKGRAY	8	Dark gray
CLR_DARKBLUE	9	Dark blue
CLR_DARKRED	10	Dark red
CLR_DARKPINK	11	Dark pink
CLR_DARKGREEN	12	Dark green
CLR_DARKCYAN	13	Dark cyan
CLR_BROWN	14	Brown
CLR_PALEGRAY	15	Pale gray

Entries after CLR_PALEGRAY have device-dependent defaults. The Presentation Manager maps the color index name to an index number, and uses the number to address the appropriate color. The index numbers for the default color table are shown in the second column of the table.

The colors produced by the values CLR_DEFAULT, CLR_BACKGROUND, and CLR_NEUTRAL in the default color table depend on the type of device receiving the output. CLR_BACKGROUND produces the natural background color for the device. This is the color of the paper on a printer, and the window background color (white, by default) on a display. CLR_NEUTRAL is the contrast color to CLR_BACKGROUND. This is usually black on a printer, and the default window text color (black, by default) on a display. CLR_DEFAULT produces the same effect as CLR_NEUTRAL.

These three indexes allow you to select colors according to their purpose, and thus build device-independence into your applications. The purpose does not vary from one device to another, although the color used to implement that purpose can. For example, if CLR_NEUTRAL is changed to blue on a particular device, your application will produce anything drawn in CLR_NEUTRAL in blue rather than black.

Defining a Logical Color Table

If you want to make any changes to the default logical color table, you use the GpiCreateLogColorTable (GSCLCT) call. Using this call, you can:

- Replace a part or the whole of the default color table
- Add color definitions to the default color table
- Reset the logical color table to its default values.

Any locally defined color table is referred to as a *loaded* color table. A loaded color table is stored in the presentation space, and is presentation-space specific. It can have either of the following two formats:

1. You can supply an array of color indexes and their associated RGB values. This is how the default logical color table is defined. If you want to add to that table or change some of its contents (rather than replace it completely), this is the format you should use. A table in this format is said to be in *index mode.*

 To alter a logical color table in index mode, specify either LCOLF_INDRGB or LCOLF_CONSECRGB as the format value on GpiCreateLogColorTable (GSCLCT).

 If you use the format LCOLF_INDRGB, you can supply an array of index and RGB pairs with which the table is to be updated. The values do not have to be consecutive. You can change the contents of the supplied table, and you can add entries to end of the table.

 If you use the format LCOLF_CONSECRGB, you supply an array of RGB values and a starting index only. The RGB values must therefore apply to consecutive entries in the table. You cannot use a single GpiCreateLogColorTable (GSCLCT) call to redefine CLR_BLUE and CLR_PINK, without also specifying an RGB value for CLR_RED. As with LCOLF_INDRGB, you can use this format to change the contents of the supplied table, and to add entries to the end of the table.
2. Because the RGB value is itself unique, you can supply an array of RGB values without index values. To do this, you specify a format of LCOLF_RGB. A table defined in this way is said to be in *RGB mode.* You can use RGB mode only when you are replacing the whole of the default logical color table.

Whichever color-table format you use, the RGB value itself is a 4-byte integer with 8 bits for each primary color, and is defined as:

```
(R * 65536) + (G * 256) + B
```

where R is the red intensity value, G is the green intensity value, and B is the blue intensity value. The maximum intensity for each primary color is 255.

If you use the color-index names to address the contents of a loaded color table in index mode, they retrieve the color identified by the associated index number. For example, the index CLR_PINK retrieves the color addressed by index number 3. CLR_BACKGROUND produces the color addressed by index number 0, and CLR_NEUTRAL produces the color addressed by index number 7. Thus, if you want CLR_NEUTRAL to be blue, define your table so that index number 7 addresses the RGB definition of the appropriate shade of blue in your table.

Color Dithering

If you define a color that is not available in the physical color table of the current device, the Presentation Manager can use a process known as *color dithering* to get a closer approximation to the color you want. The color-dithering process takes advantage of the way in which the human eye interprets color: if you set every other pel to one color, and all the intermediate pels to a different color, together they produce the effect of a third color at normal viewing distances. This is analogous to the way in which the red, green, and blue color guns illuminate the phosphors in a single pel to produce the effect of a distinct color. The 'checkerboard' effect is just one of the ways in which color dithering can be implemented. Color dithering works only when you are producing a solid mass of color, such as an area-fill pattern. It does not have the desired effect on line primitives.

When you want to use only the distinct colors defined in the physical color table, you can prevent color dithering by setting the LCOL_PURECOLOR value on the GpiCreateLogColorTable (GSCLCT) call. When LCOL_PURECOLOR is set, the nearest color to the one you select is used.

Realizing a Logical Color Table

A logical color table can be loaded into the physical color table of the currently associated device, so that the indexes (and the colors they retrieve) are the same for both tables. This process is known as *realization*. Realization of the color table causes the closest possible match between the colors you have defined and the colors that are available on the device, but it works only if the logical table fits into the physical table.

If the logical color table is too large, the indexes are mapped from the logical to the physical table as if the table had been realized. That is, index number 1 in the logical color table maps to the first entry in the physical color table, index number 2 maps to the second entry, and so on. This does not give predictable results. Therefore you should compare the sizes of the two tables using the DevQueryCaps (WDQCAP) call (for the physical table) and the GpiQueryColorData (GSQCOD) call (for the logical table) before you realize the logical table.

If a logical color table is to be realized, you must use the LCOL_REALIZABLE option when you define the color table with the GpiCreateLogColorTable (GSCLCT) call. Use the GpiRealizeColorTable (GSRCT) call when you are ready to realize the table. This call fails if you do not use the LCOL_REALIZABLE option when you define the color table. If you do not issue GpiRealizeColorTable (GSRCT) when you have defined the table as realizable, the indexes are mapped from the logical to the physical table as they are when the logical table is too large for the physical table.

You should not realize a color table if the current output device is a screen window, unless the window is maximized. The appearance of other visible applications is spoiled if you realize a logical color table without maximizing the window. Similarly, if you realize a color table, you must reverse the process using the GpiUnrealizeColorTable (GSUCT) call before the size of the window is reduced.

Not all device drivers support realization of the color table. If you issue GpiCreateLogColorTable (GSCLCT) with the LCOL_REALIZABLE option when the current device does not support realization, a warning is issued, but no error is raised. The color table is treated as unrealized while the presentation space is associated with a device that does not support realization. If you subsequently associate the presentation space with a device that does support realization, the LCOL_REALIZABLE option is honored. You do not have to redefine the logical color table.

However, if you issue the GpiRealizeColorTable (GSRCT) call for a device that does not support realization, an error condition is raised. GpiRealizeColorTable is a device-specific call, and has to be issued each time you reassociate the presentation space if you want the color table to be realized. You can use the DevQueryCaps (WDQCAP) call to determine the color-table support available on the current device.

Finding Out About the Available Colors

Several query calls have been provided so that you can get information about the current logical and physical color tables before defining a logical color table.

The GpiQueryColorData (GSQCOD) call tells you whether the default logical color table is in effect. If it is not, the following information about the loaded logical color table is returned:

- The format of the current logical color table.
- The smallest and largest color indexes (if the table is in index mode) supported in this color table. The smallest color index is always zero, and the largest color index is never less than 15, because you cannot delete entries from the default logical color table.

The GpiQueryRealColors (GSQRC) call returns to you the RGB values of each of the distinct colors defined in the physical color table of the current device. You can also request the index in the current logical color table that retrieves each of the colors in the physical table.

As colors vary from device to device, you can find out the nearest color to a specified color that is available on a particular device. As input to the GpiQueryNearestColor (GSQNC) call you supply the RGB values of the color you want. Output from this call is the RGB value of the nearest color available in the physical color table of the associated device.

You can discover the RGB value of a particular color using the GpiQueryRGBColor (GSQRGB) call. As input to this call you supply the index to the logical color table entry of the color you are interested in. Output from the call is the RGB value that the index would retrieve from the physical color table. If you have loaded a logical color table in RGB mode, this function is the same as that provided by GpiQueryNearestColor (GSQNC).

Conversely, if you have the RGB value and want to know the index value that retrieves that RGB color (or the closest match to that color) from the physical color table, use the GpiQueryColorIndex (GSQCOI) call.

The Color Attribute

The Presentation Manager's graphic interface uses a variety of colors. For example, the background of a window is a different color from the window border, and active windows can be differentiated from inactive windows by their colors. These colors are referred to as the *system colors*, and they are defined in the system color table, which is separate from the application's logical color table. You can find out the RGB values of the system colors using the WinQuerySysColor (WIQSCO) call. These colors can be changed interactively using the Screen Colors window in the Control Panel. For more information about the system colors, see "Using the System Colors" on page 14-7.

The colors of graphic primitives are specified separately from the system colors. Every primitive has a *foreground* color. Some also have a *background* color.

The Foreground of a Primitive

The foreground of a primitive is the primitive itself. For example, the foreground of a full arc primitive is the full arc, as shown in Figure 21-2.

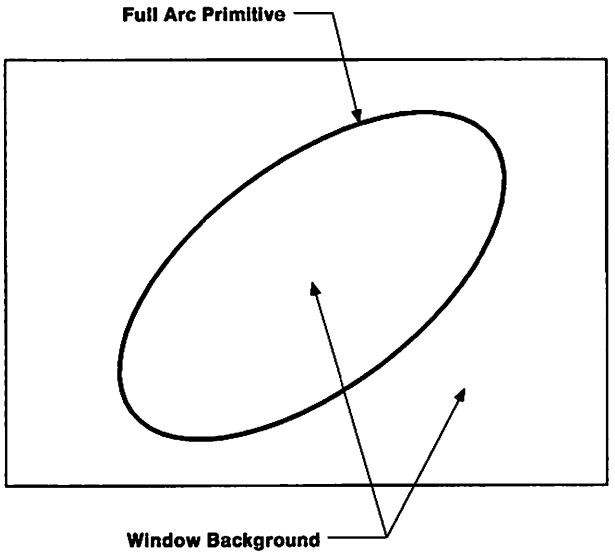


Figure 21-2. The Foreground of a Primitive. The full arc primitive is drawn in a different color from the window background.

By default, the foreground color of all primitives is that addressed by the index CLR_DEFAULT. In the supplied color table, this produces black on a graphics display. If you replace the default logical color table, CLR_DEFAULT produces the color addressed by index number −3.

The Background of a Primitive

Primitives that have a background are areas, character strings, images, and markers. The background of any character or marker primitive, whether the primitive is from an image or an outline font, is the *whole* character or marker box. That is, the background is drawn first, and the foreground is then drawn on top of it. Similarly, the

background of the area primitive is the whole of the area to be filled. The background of the image primitive, however, is that part of the primitive in which the pels are not set. Figure 21-3 shows the background of a primitive.

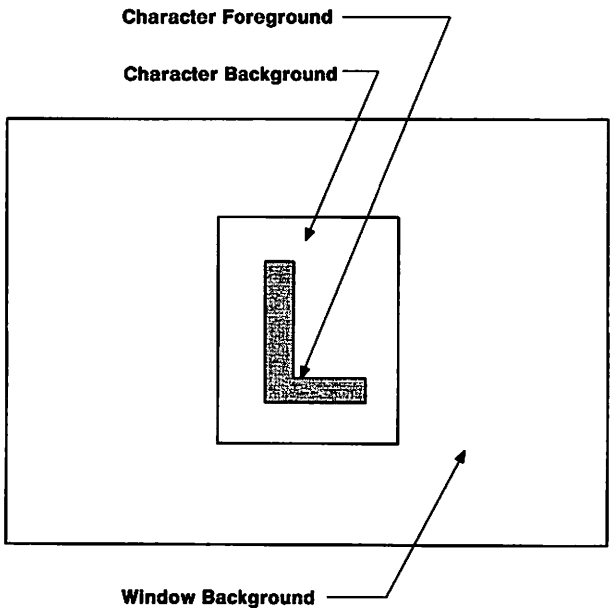


Figure 21-3. The Background of a Primitive. The background of the character-string primitive is the whole character box. The character foreground is drawn on top of the background.

The index to the default background color is CLR_BACKGROUND (index 0), which provides a background color that is appropriate for the device. On a printer, CLR_BACKGROUND is the color of the paper. On a screen device, CLR_BACKGROUND is the default window background color.

Changing the Foreground and Background Colors of Primitives

To change the current foreground color, you use the GpiSetColor (GSCOL) call. As input to this call, you supply either the index to the required color in the current logical color table or the RGB value of the color, depending on the format of the table. Color indexes higher than those supplied in the default color table must be explicitly loaded before they can be used.[1]

The color you specify becomes the current color, and the foreground of any primitive drawn subsequently is drawn in that color. You can also set the current foreground color for a particular primitive type using the GpiSetAttrs (GIATR) call. For example, if you set the current foreground color to CLR_RED using the GpiSetColor (GSCOL) call, and set the current foreground color for marker primitives to CLR_CYAN using the GpiSetAttrs (GIATR) call, all subsequent markers are cyan, and all other primitives are red.

To change the current primitive-background color, you use the GpiSetBackColor (GSBCOL) call. If you want this to be different from the background color of the output area, so that the whole of the defined primitive is visible, make sure that the current window background color is not the same as the current primitive background color. You should also select an appropriate background-mix mode. Mix modes are described below.

The Mix Attribute

The mix attribute determines how each primitive you draw is combined with any existing drawing. In color applications, it determines the color that results when one primitive is drawn on top of another. There are two forms of the mix attribute: *foreground mix* and *background mix*.

The foreground-mix attribute governs how the foreground of a primitive is combined with the existing drawing, and it applies to all primitive types. The background-mix attribute governs how the background of a primitive is combined with the existing drawing, and is applicable only to those primitives that have a background. Primitives that can be affected by the background-mix attribute are areas, character strings, images, and markers.

[1] It is also possible to specify one of the system colors (for example, SYSCLR_ACTIVEBORDER) as the current foreground (or background) color of a primitive. The color is as defined in the system color table, and the logical color table is ignored.

The available foreground-mix settings are listed in Table 21-2.

Table 21-2. Foreground-Mix Settings	
Mix Value	**Effect**
FM_DEFAULT	Default (overpaint)
FM_OR	OR
FM_OVERPAINT	Overpaint
FM_XOR	Exclusive-OR
FM_LEAVEALONE	Leave-alone (invisible)
FM_AND	AND
FM_SUBTRACT	(inverse source) AND dest.
FM_MASKSRCNOT	Source AND (inverse dest.)
FM_ZERO	All zeros
FM_NOTMERGESRC	Inverse (source OR dest.)
FM_NOTXORSRC	Inverse (source exclusive-OR dest.)
FM_INVERT	Inverse (dest.)
FM_MERGESRCNOT	Source OR (inverse dest.)
FM_NOTCOPYSRC	Inverse (source)
FM_MERGENOTSRC	(inverse source) OR dest.
FM_NOTMASKSRC	Inverse (source AND dest.)
FM_ONE	All ones.

The colors you get from these mix modes result from Boolean operations on the RGB values of the color of the original drawing, and the color of the new primitive. The RGB values are those from the *physical* color table. Unless you have realized the logical color table, the results of most of these mixes are unpredictable in color applications.

The first five of the foreground mix modes are also available as background mix modes. The background mix modes are listed in Table 21-3.

Table 21-3. Background-Mix Settings	
Mix Value	**Effect**
BM_DEFAULT	Default (leave-alone)
BM_OR	OR
BM_OVERPAINT	Overpaint
BM_XOR	Exclusive-OR

Table 21-3. Background-Mix Settings	
Mix Value	**Effect**
BM_LEAVEALONE	Leave-alone (invisible)

The most frequently used foreground-mix modes are FM_OVERPAINT, which is the default value, FM_OR, and FM_XOR. The most frequently used background-mix modes are BM_LEAVEALONE, which is the default value, and BM_OVERPAINT. These mix modes are described below.

Overpaint Mix Mode

In FM_OVERPAINT mode, the foreground of the primitive replaces any existing drawing that is in the same area of the presentation page. If the existing drawing is in yellow, for example, and the new drawing is in red, at the points of overlap the drawing is red. This is the default foreground mix mode. Because one color is replacing another and no color mixing is being performed, the effects of overpaint mix mode are entirely predictable. The effects of FM_OVERPAINT mode are shown in Figure 21-4.

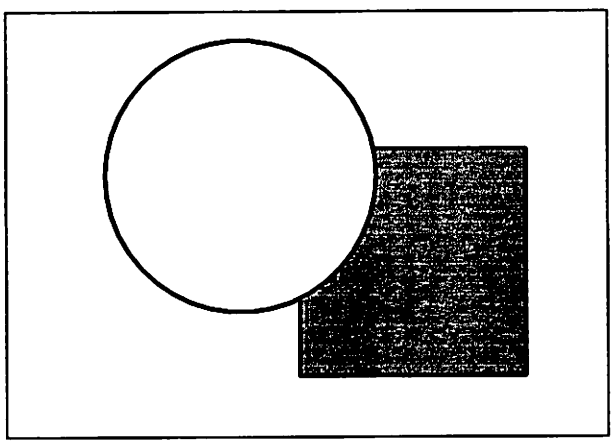


Figure 21-4. Overpaint Foreground-Mix Mode. The circle is drawn on top of the square. At the points of overlap, the output is the color of the circle.

If the primitive has a background (and it is not an image primitive), this mix mode causes the foreground of the primitive to be drawn on top of the primitive background. No color mixing is performed.

In BM_OVERPAINT mode, the primitive background replaces the existing drawing, as shown in Figure 21-5.

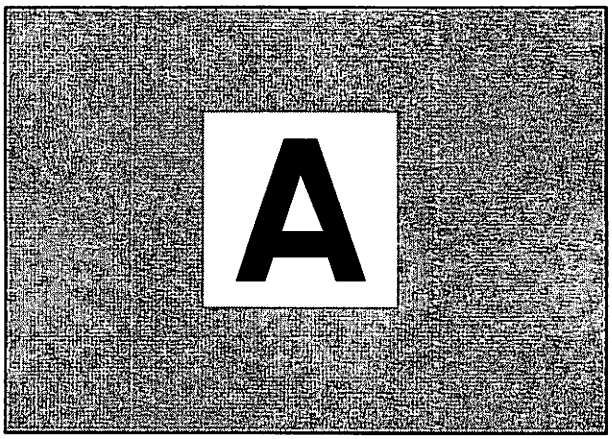


Figure 21-5. Overpaint Background-Mix Mode. In BM_OVERPAINT mode, the background of the primitive is apparent only if it is drawn in a different color from the output-area background. Note that, in this example, the foreground-mix mode is FM_OVERPAINT.

OR Mix Mode

In FM_OR mode, the foreground of the new primitive is merged at the points of overlap with the existing drawing. This is effected by ORing together the RGB values (from the *physical* color table) of the overlapping colors to produce a third color. The resulting color is unpredictable if the logical color table has not been realized. OR mix mode is useful when you want to make the common points of two graphics distinct from those points that belong to one of the graphics only. Its effects are shown in Figure 21-6.

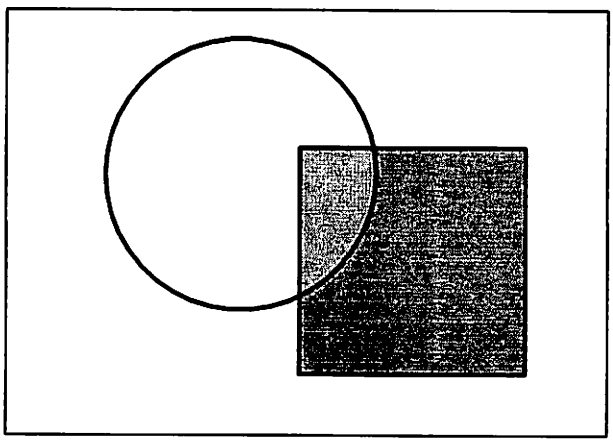


Figure 21-6. OR Mix Mode. The circle is drawn on top of the square. At the points of overlap, the RGB values of the circle and square colors are ORed together to produce a third color.

In BM_OR mode, the background of the primitive is merged with the existing drawing according to the same rules that apply in FM_OR mode.

Exclusive-OR Mix Mode

In FM_XOR mode, graphics are not drawn at points of overlap. Exclusive-OR mode is available on display devices only, and is used for graphics animation, when you must be able to move an individual graphic without destroying the graphics around and underneath it. Typically, you:

1. Draw the graphics object in exclusive-OR mode.
2. Calculate the object's next position on receipt of WM_BUTTON1DOWN, for example.
3. Draw the object again in its current position, still in exclusive-OR mode. This effectively erases it from its current position without destroying graphics with which it overlaps.
4. Draw the object in its new position.

You can automate this sequence to some extent by defining specific graphics objects as *dynamic*. Graphics that you define as dynamic are always drawn in exclusive-OR mode, regardless of the current mix modes. Dynamic graphics are described in Chapter 22, "Graphics Segments."

The effects of FM_XOR mode are shown in Figure 21-7.

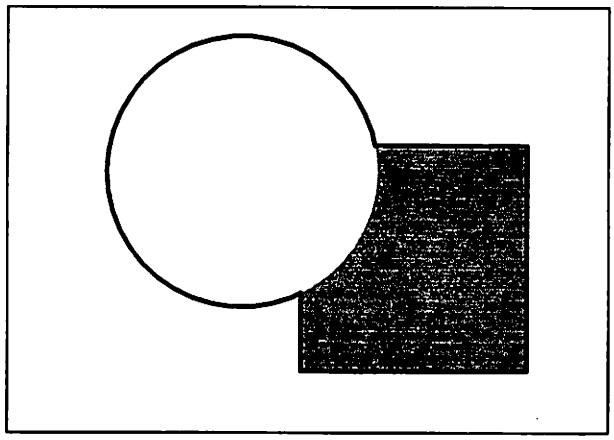


Figure 21-7. Exclusive-OR Mix Mode. The circle is drawn on top of the square. At the points of overlap, graphics are not drawn.

In BM_XOR mode, the background of the primitive is merged with the existing picture according to the same rules that apply in FM_XOR mode.

Leave-Alone Mix Mode

In FM_LEAVEALONE mode, the foreground of the primitive is not drawn. Leave-alone mode is most often used as a background-mix mode. In BM_LEAVEALONE mode, the background of the primitive is not drawn, as shown in Figure 21-8.

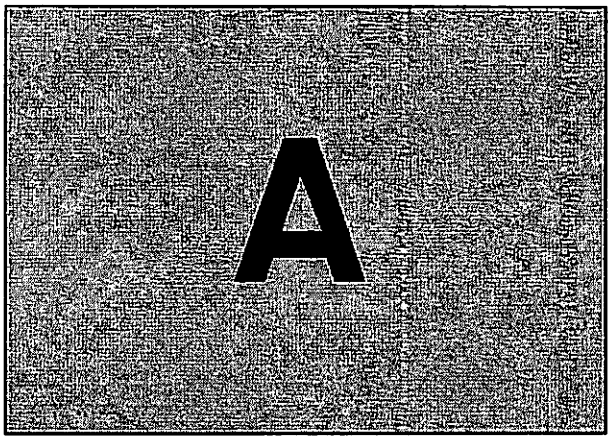


Figure 21-8. Leave-Alone Background-Mix Mode. This is the background-mix mode that is most often used for character strings and marker primitives. It is generally used with a foreground-mix value of FM_OVERPAINT.

Specifying Foreground- and Background-Mix Modes

You specify the current foreground-mix value for all primitives using the GpiSetMix (GSMIX) call. To discover the current foreground-mix setting, you use the GpiQueryMix (GSQMIX) call, which returns to you the current foreground-mix setting for the line primitive.

To specify a background-mix value for all primitives, you use the GpiSetBackMix (GSBMIX) call. The GpiQueryBackMix (GSQBMX) call returns the current background-mix value for the character-string primitive. You can also specify the current foreground- and background-mix values for a single primitive type using the GpiSetAttrs (GIATR) call.

Note: Not all devices support all of the background and foreground mix modes described. When a device does not support the mix mode you choose, the default mix mode is used instead. No error condition is raised. You can use the DevQueryCaps (WDQCAP) call to determine whether a mix mode is supported on a specific device.

Drawing Color Graphics on Monochrome Devices

When you map color graphics to a monochrome device (screen, printer, or bit map), that device is said to have a *reset* color and a *contrast* color. The reset color is the color the GpiErase (GSERSE) call clears to, and is either black or white. If the reset color is black, the contrast color is white, and if the reset color is white, the contrast color is black.

To determine whether the reset color is to be black or white, the color retrieved by the index CLR_BACKGROUND from the current logical color table is examined. If you have not loaded a logical color table, CLR_BACKGROUND is either

- The paper color on a printer, or
- SYSCLR_WINDOW on a display.

If you have loaded a logical color table, CLR_BACKGROUND is the color addressed by color index 0 on any device.

If the color retrieved by CLR_BACKGROUND is either white or black, that color becomes the reset color. For example, if you are drawing to a screen, have not loaded a color table, and the system colors have not been altered, the background color is white, because SYSCLR_WINDOW produces a white background by default. In this instance, the contrast color would be black. If the color retrieved by CLR_BACKGROUND is neither white nor black, the color translates to whichever of black and white it is nearest to. As a rule, dark colors produce a black background, and pale colors produce a white background. For example, if you are using a loaded color table, or if the color retrieved by SYSCLR_WINDOW has been altered (either interactively or by WinSetSysColors), CLR_BACKGROUND could be dark gray. In this instance, the reset color would be black, and the contrast color would be white.

When the reset color has been established, the Presentation Manager applies the following rules when mapping color graphics to a monochrome device:

- Any color graphics that are drawn in CLR_BACKGROUND, and any graphics that are defined in the actual reset color (which will be either black or white) are drawn in the reset color. Any other graphics are drawn in the contrast color.

- The index CLR_WHITE produces white, and the index CLR_BLACK produces black, regardless of whether the reset color is black or white.
- If no color table has been loaded and CLR_DEFAULT or CLR_NEUTRAL are used as the foreground color, they produce the contrast color. If used as the background color, they produce the reset color.
- If you issue GpiQueryNearestColor (GSQNC) to a monochrome device, the following occurs:
 - If the color you supply on input is the reset color, the reset color is returned to you
 - If the color you supply on input is not the reset color, the contrast color is returned to you.

Drawing Color Area-Fill Patterns on Monochrome Devices

An area primitive is drawn according to the current foreground- and background-mix modes, and in the current area-foreground and area-background colors.

When you draw a monochrome pattern on a color device, the 1 bits of the pattern translate to the current area-foreground color, and the 0 bits translate to the current area-background color. When you draw a color pattern on a monochrome device, and if the current pattern is anything other than PATSYM_DEFAULT or PATSYM_SOLID from the default pattern set, a decision has to be made about which color should translate to 1 bits, and which color should translate to 0 bits. If you draw a pattern of diagonal lines in which the foreground color is red and the background color is cyan, for example, cyan would be translated to white (1 bits) because cyan is closer to white. Red would therefore be translated to black (0 bits). The effect of translating this color pattern to a monochrome surface is summarized below:

Pattern	As 1s and 0s	Color Surface	Monochrome Surface
\ \	10001000	RcccRccc	01110111
\ \	01000100	cRcccRcc	10111011
\ \	00100010	ccRcccRc	11011101
\ \	00010001	cccRcccR	11101110
\ \	10001000	RcccRccc	01110111
\ \	01000100	cRcccRcc	10111011
\ \	00100010	ccRcccRc	11011101
\ \	00010001	cccRcccR	11101110

The original pattern of 1s and 0s is used, however, when deciding which part of the pattern is the background, and which part is the foreground. Thus, if the background-mix mode is BM_LEAVEALONE, the following occurs:

Pattern	As 1s and 0s	Color Surface	Monochrome Surface
\ \	10001000	R...R...	0...0...
\ \	01000100	.R...R..	.0...0..
\ \	00100010	..R...R.	..0...0.
\ \	00010001	...R...R	...0...0
\ \	10001000	R...R...	0...0...
\ \	01000100	.R...R..	.0...0..
\ \	00100010	..R...R.	..0...0.
\ \	00010001	...R...R	...0...0

The 1 bits on the monochrome surface are still interpreted as the background of the primitive, and are not drawn in BM_LEAVEALONE mode.

When a bit map is used as an area-fill pattern, any bit drawn in the current area-background color is set to 0, and all other bits are set to 1 on a monochrome surface. Thus, if the current area-background color is blue, all blue bits in the bit map are set to 0, and all other bits are set to 1. The 0 bits constitute the background of the primitive.

If the pattern is solid (PATSYM_DEFAULT or PATSYM_SOLID from the supplied pattern set), the following occurs:

- If color dithering is switched off and you are drawing a color pattern to a color surface, the color nearest to the one you specify is used.
- If color dithering is switched on and you are drawing a color pattern to a color surface, a combination of colors may be used to achieve the effect of the requested color. For example, if you choose pink on a surface where pink is not available, a combination of red and white pels might be used to achieve the effect of the color.
- If color dithering is switched on and you are drawing a color pattern to a monochrome surface, sufficient pels are set to suggest the intensity of the requested color.

Dithering can be enabled and disabled using the LCOL_PURECOLOR option of GpiCreateLogColorTable (GSCLCT).

Chapter 22. Graphics Segments

A *graphics segment* is primarily a means of grouping together and storing graphic primitives and their attributes. Although the graphics within a segment are usually related to each other in some way, they do not have to be. For example, a graphics segment could contain the definition of a discrete object, such as the wheel of a car or the window of a house, that you want to manipulate as a single graphics object. Equally, it could contain a number of unrelated GPI calls that you want to keep and execute at specific times.

There are many advantages to using graphics segments, and you will find that the more powerful and useful graphics functions are available to you only when you store your graphics in segments. Probably the most important advantage of using graphics segments is that they allow you to retain graphics in the *segment store* of the presentation space for later use. Graphics in the segment store can be drawn any number of times to any number of device contexts. Segmentation alone is not enough to cause graphics to be retained. Before you create a segment that you want to keep in segment store (a *retained segment*), you have to select an appropriate *drawing mode*.

The Drawing Mode

The Presentation Manager has three drawing modes. These are:

- Draw mode
- Retain mode
- Draw-and-retain mode.

Draw Mode

In draw mode, graphics are output immediately to the currently associated device and are not retained in segment store. When they have been drawn, they cannot be used again unless they are recreated. For example, when a window containing draw-mode graphics is moved or sized, the graphics have to be recreated by the application. Draw-mode graphics are suitable if an application is creating fairly simple graphics quickly, or if an application is maintaining its own graphics database. In the latter case, there is nothing to gain by retaining graphics.

Retain Mode

In retain mode, graphics are retained in the segment store only. They are not directed to the current device as they are created. (In retain mode, the presentation space does not have to be associated with a device context when graphics are being defined.) Many of the GPI query calls that return current attribute values are invalid in retain mode. This is because current attribute values have no effect when graphics are not being sent to an output device.

Draw-and-Retain Mode

In draw-and-retain mode, graphics are both drawn on the current device as they are created, and stored for later use. The GPI query calls that return current attribute values are valid in draw-and-retain mode.

Specifying the Drawing Mode

To specify the current drawing mode, you use the GpiSetDrawingMode (GSDM) call, on which you specify one of the following values:

- DM_DRAW
- DM_RETAIN
- DM_DRAWANDRETAIN.

The drawing mode that you select becomes current for the presentation space, and it can be changed any number of times. You select the appropriate drawing mode before starting to create a segment. The default drawing mode is DM_DRAW.

The drawing mode (as defined by the GpiSetDrawingMode parameter), in conjunction with the context (inside or outside a graphics segment), produces the *actual drawing mode*. It is the actual drawing mode that determines whether graphics are drawn directly to the output device, retained in segment store, or both drawn and retained. How the actual drawing mode is arrived at is summarized in Table 22-1 on page 22-2.

Table 22-1. The Current Drawing Mode			
GpiSetDrawingMode parameter	**Context**		
	Chained Segment	**Unchained Segment**	**Outside Segment**
DM_DRAWANDRETAIN	draw-and-retain	retain	draw
DM_RETAIN	retain	retain	draw
DM_DRAW	draw	retain	draw

For example, if you are drawing graphics outside a segment when the current drawing-mode parameter is DM_RETAIN, the actual drawing mode is 'draw'.

Graphics segments can be chained together so that they can be handled as a group. The chaining of graphics segments is described on page 22-6. Whether a segment is chained, in conjunction with the drawing mode, determines whether the segment is retained. As you can see from Table 22-1, graphics in unchained segments are always retained, regardless of the current drawing-mode parameter. Similarly, graphics outside segments are always drawn without being retained. Graphics within chained segments, however, always conform to the current drawing mode parameter. That is, they are drawn in DM_DRAW mode, retained in DM_RETAIN mode, and both drawn and retained in DM_DRAWANDRETAIN mode.

To summarize:

- Any graphic that you want to use more than once must be within a segment.
- You cannot retain segments created when the current drawing-mode parameter is DM_DRAW, unless they are unchained segments.
- You cannot retain primitives outside segments, regardless of the current drawing mode.

Note: A micro presentation space has no segment store. You cannot, therefore, create graphics segments in a micro presentation space, nor can you change the drawing mode, which is always DM_DRAW.

Creating a Graphics Segment

To signal the start of a graphics segment, you use the GpiOpenSegment (GSSEG) call, which is valid only in a normal presentation space.

You have to give each segment a name. If you want to refer to the segment individually in later graphics operations (for example, if you want to edit the segment), the name you supply must be greater than zero and it must be unique within the presentation space. If you do not want to refer to the segment individually after it has been created, you can give the segment a name of zero. You might do this when you are creating nonretained segments, for example. Any number of segments can have the name zero. They are referred to throughout this book as *zero segments*. To find out which names have already been used in a presentation space, use the GpiQuerySegmentNames (GSQNAM) call. This call returns to you the names of all retained segments, excluding zero segments, within a specified name range.

In general, the graphics orders generated from the GPI calls that follow the GpiOpenSegment (GSSEG) request become a part of a retained segment, and are executed every time the segment itself is drawn. However, not all GPI calls cause graphics orders to be added to the current segment. As a general rule, each call that writes a graphics order to the current segment includes the appropriate graphics order code in its description in the *Programming Reference: Volume 2*. For example, the description of the GpiPolyLine (GIPLNE) call includes the order code OCODE_GCLINE. GpiPolyLine therefore writes a graphics order to the current segment. There are a few GPI calls that you cannot issue while there is an open segment (the GpiOpenSegment call itself is an example of this) and you can find these calls identified in Appendix B, "GPI Calls."

When you have finished creating a graphics segment, you close it using the GpiCloseSegment (GSSCLS) call. There can be only one open segment at a time in a single graphics presentation space, so you must close one segment before going on to the next.

Each segment can be considered as a subset of the presentation space. For example, the current position and attribute values that apply before you issue the GpiOpenSegment (GSSEG) call cannot be guaranteed to be in effect after you have issued GpiOpenSegment. You are likely, therefore, to begin each segment with a number of attribute-setting requests and, possibly, with a GpiSetCurrentPosition (GICP) call. Similarly, there is some degree of clean-up processing associated with the GpiCloseSegment (GSSCLS) call (also known as *close-segment processing*) that can make current attribute values unreliable. For more information about the effects of GpiOpenSegment and GpiCloseSegment on current attributes, see Appendix C, "Graphics Attributes."

There can be up to 16 378 segments in the segment store of a single presentation space, though the size of any individual segment is limited only by the amount of storage available to you.

The Elements of a Segment

A graphics segment is bracketed by the GpiOpenSegment (GSSEG) and GpiCloseSegment (GSSCLS) calls. The contents of a segment are typically the graphics orders generated by the GPI calls that create graphics primitives, such as lines or markers, and that assign attributes to those primitives, such as color or line type. Generally, each of the GPI calls within the open and close segment bracket generates one *element* of the segment. This section describes how to address individual elements of a segment, how to add elements to a new segment, how to classify elements, and how to put labels in a segment. You need to understand these activities when you are creating graphics segments, but their value is more apparent when you are editing those segments after they have been closed. How to edit a graphics segment is described on page 22-4.

The Element Pointer

Each element of a segment can be addressed by the *element pointer*. Whenever you issue the GpiOpenSegment (GSSEG) call, the element pointer is set to 0. The first element that you add to a newly created segment causes the element pointer to be set to 1, and each subsequent element causes the pointer to be incremented by 1. When a segment is closed, the element pointer position is always reset to 0.

You can explicitly set the element pointer to a specified value using the GpiSetElementPointer (GSEP) call. For example, to address the sixth element in a segment, you set the element pointer to 6. You can also move the element pointer by a specific number, rather than to a specific number, using the GpiOffsetElementPointer (GSOEP) call. If the element pointer currently addresses element 3 and you want to move it to element 8, you can code an offset of 5 on GpiOffsetElementPointer. You can specify a negative offset value, so that the pointer moves backward rather than forward through the segment. To request the contents of a part or all the element at the current pointer position, use the GpiQueryElement call.

Adding Elements to a New Segment

There are four ways of adding elements to a newly created segment:

- The simplest way of supplying graphics data to a segment is to issue one or more GPI calls. Each GPI call you issue generates one element of the segment.
- You can signal the start of a new element by issuing a GpiBeginElement (GSELEM) call, and follow it with one or more GPI calls. You close the element by issuing a GpiEndElement (GSENDE) call. All the GPI calls within the GpiBeginElement and GpiEndElement bracket generate a single element of the segment, and they therefore occupy a single element-pointer position. You cannot subsequently access the graphics orders individually using the element pointer, although the orders can be accessed directly. For more information about accessing graphics orders, see Appendix D, "Graphics Orders."

 The purpose of GpiBeginElement (GSELEM) and GpiEndElement (GSENDE) is to allow you to add to a segment a single element that comprises more than one graphics order. You are likely to do this when you know that, at a later stage, you want to retrieve those orders and manipulate them as a group rather than as individual calls. There is, therefore, no reason for using these two calls to enclose a single GPI call and, although they are valid in any drawing mode, you are unlikely to use them in draw mode.

 Elements cannot be nested, so you cannot begin two elements in succession without an inter-

vening GpiEndElement (GSENDE) request. Some GPI calls cannot be issued between the GpiBeginElement and GpiEndElement calls. For details, see Appendix B, "GPI Calls."

- You can use the GpiElement call. As input to this call you supply the graphics orders themselves, rather than the GPI calls that generate them. However much data you supply on this request, it is considered a single element of the segment. Multiple orders added to a segment in this way cannot be accessed individually using the element pointer, but they can be accessed directly. For more information about accessing graphics orders, see Appendix D, "Graphics Orders." You cannot include a GpiElement call within a GpiBeginElement and GpiEndElement bracket, and the data passed on the GpiElement call must not include the GpiBeginElement and GpiEndElement calls.
- You can use the GpiPutData (GSPUT) call. As with the GpiElement call, you supply graphics orders on input. You can even include GpiBeginElement (GSELEM) and GpiEndElement (GSENDE) orders in this data, and thus add more than one element to the segment. GpiPutData (GSPUT) is useful when you have a large amount of data to add to a segment. There is also a GpiGetData (GSGET) call, which copies data from a segment to application storage.

Element Types

Each element that results from a single GPI call has an associated *element type* value. The element type is generally the graphics-order code associated with the graphics order that is generated by the GPI call, and is supplied by the system. For example, the GpiLine (GILINE) call has the element-type value OCODE_GCLINE, which identifies it as a line-drawing request. You can find a complete list of the graphics-order codes in the *Programming Reference: Volume 2.*

When you use the GpiElement call or the GpiBeginElement (GSELEM) call, you can add several graphics orders to a segment. Those orders become one element of the segment. You can supply your own classification for these compound elements by supplying a value for the type parameter of GpiElement and GpiBeginElement (GSELEM). For example, the orders passed on a single GpiElement call could change the line type to dotted, set the current color to blue, and draw a line. You could classify this element with a type value that identifies it as a blue dotted line. To discover the current classification of an element, use the GpiQueryElementType (GSQETS) call.

Labels

You can include *labels* in a segment to make locating particular elements of the segment easier. A label is just a place-holder or reference point, and is itself an element of the segment. To include a label in a segment you use the GpiLabel (GSINLB) call. You set the element pointer to the label position using the GpiSetElementPointerAtLabel (GSEPLB) call, and increment the pointer position to locate the element itself using the GpiOffsetElementPointer (GSOEP) call.

Labels do not have to be unique. If you use nonunique labels, the element pointer positions itself at the first occurrence of the label, starting from the current pointer position. If the label is not found between the current pointer position and the end of the segment, an error condition is raised.

Editing a Segment

You can edit the contents of any retained segment that has a unique name. You can also edit the contents of a retained zero segment that has not yet been closed. Zero segments cannot be edited after they have been closed, as you cannot refer to a zero segment when it is no longer the current open segment.

Before you can start the edit, you must set the current drawing-mode parameter to DM_RETAIN. (You cannot edit a segment if the current drawing-mode parameter is DM_DRAWANDRETAIN or DM_DRAW.) To begin the edit, you issue the GpiOpenSegment (GSSEG) call and specify the name of the segment you want to edit. When you have finished editing a segment, you close it by issuing GpiCloseSegment (GSSCLS).

Inserting or Replacing Data in a Graphics Segment

The Presentation Manager provides two *edit modes*, which are SEGEM_INSERT (insert) and SEGEM_REPLACE (replace). You can insert new elements following the element at the current pointer position, or you can replace the element

that is at the current pointer position. To select an edit mode for a presentation space, use the GpiSetEditMode (GSEDM) call. The edit mode applies to all segments in the presentation space until you next change it. The edit mode is not an attribute of a particular segment and can be changed at any time. The default edit mode, which is set when you create a presentation space, is SEGEM_INSERT. (When you create a graphics segment, you are editing it in SEGEM_INSERT mode.) To find out what the current edit mode is, use the GpiQueryEditMode (GSQEDM) call.

You can insert or replace data in an existing segment using any of the calls described in "Adding Elements to a New Segment" on page 22-3, except that the GpiPutData call is valid only in SEGEM_INSERT mode, and only when the element pointer is addressing the final element in the segment. (That is, when you are editing, you can use the GpiPutData (GSPUT) call only to add data to the end of a segment.)

A replace request is not valid when the element pointer is set to zero. If you issue the GpiOpenSegment (GSSEG) call to create a new segment without first ensuring that the current edit mode is SEGEM_INSERT, your first attempt to add an element to the segment may cause an error condition to be raised.

Locating Elements

When you want to change the contents of an existing segment, you use the element pointer to address the elements you are interested in. Using labels as a navigation aid may be quicker than scanning the segment for a particular element. If you label an element that is likely to be edited, you can access that element when necessary, regardless of whether its actual position in the segment has changed. If you do not use labels, you have to locate an element before you can change it.

This example shows a typical sequence of calls to change a color setting in a segment.

```
GpiSetDrawingMode
  /* Set the drawing mode to DM_RETAIN     */
GpiSetEditMode
  /* Set the edit mode to SEGEM_REPLACE    */
GpiOpenSegment
  /* Open the segment to be edited         */
GpiSetElementPointerAtLabel
  /* Locate the identifying label          */
GpiOffsetElementPointer
  /* Increment the element pointer by 1    */
GpiSetColor
  /* Replace the element at the current    */
  /* pointer position with this color-     */
  /* setting request                       */
GpiCloseSegment
  /* Close the segment                     */
```

Deleting a Part or the Whole of a Graphics Segment

You can remove a particular element or a range of elements from a named, open segment using the GpiDeleteElement (GSDLE), the GpiDeleteElementRange (GSDLER), and the GpiDeleteElementsBetweenLabels (GSDLEB) calls. When you remove an element from a segment, the element pointer addresses the element immediately before the one you deleted. As for other editing functions, the current drawing-mode parameter must be DM_RETAIN when you are deleting elements. The current editing mode (insert or replace) has no effect on the delete functions.

To delete a whole retained segment, you use the GpiDeleteSegment (GSSDEL) call. You can also delete a range of retained segments using the GpiDeleteSegments (GSDELS) call. In both instances, the segments you are deleting must have unique names. Retained zero segments cannot be deleted, and disappear only when their associated presentation space is deleted, or when it is reset. The presentation space can be reset either by the GpiResetPS (GSRPS) call with the GRES_SEGMENTS flag or the GRES_ALL flag set, or by the GpiSetPS (GIPS) call. Resetting the presentation space is described in "Reusing the Presentation Space" on page 22-11.

Copying a Segment

To copy the contents of one segment to another, you use the GpiGetData (GSGET) call. GpiGetData copies a buffer of graphics orders from a named segment (which must not be open when you issue this call) into an area of application storage, whose size you specify on input. Data retrieval has to start at the beginning of the segment. Output from GpiGetData is a count of

the number of bytes of data returned to you from the segment. If you want to retrieve the whole segment, issue GpiGetData (GSGET) until the count value is less than length of the application storage area.

The segment into which you are copying the data can be open when you issue GpiGetData (GSGET). Therefore, as each buffer of data is retrieved, you can write it out to the second segment using the GpiPutData (GSPUT) call. The current drawing mode determines whether the graphics orders are executed, stored in the segment, or both. When you issue GpiPutData, the current edit mode must be SEGEM_INSERT and, if there is data already in the segment before you issue GpiPutData, the element pointer must address the last element of the segment.

Because the Presentation Manager does not support explicit renaming of segments, this is also the method you use to rename a segment. That is, you create a second segment, copy the contents of the first segment to it, and then delete the original segment.

The Attributes of a Segment

Each segment, whether retained or nonretained, has a number of attributes that you must give some consideration to whenever you create a new segment. The attributes of a segment are quite different from the attributes of a graphics primitive. Segment attributes have on and off settings, and each has a default value that applies unless you change it. The attributes of a segment are:

- The *chained* attribute
- The *fast-chaining* attribute
- The *dynamic* attribute
- The *detectability* attribute
- The *propagate detectability* attribute
- The *visibility* attribute
- The *propagate visibility* attribute.

The Chained Attribute

Segments can be chained together so that they can be manipulated as a group. When you define a segment with the chained attribute switched on, that segment becomes a part of the *segment chain.* The segment chain comprises all chained segments defined in a single presentation space. There can be only one segment chain at a time in a single presentation space, and all chained segments are chained to each other in the order in which you create them. Zero segments must have the chained attribute, and segments are defined as chained by default.

There is usually a logical relationship between the segments in a segment chain, though this is not a requirement. The whole segment chain can be drawn using the GpiDrawChain call (GSDCHN). (You can find more information about the GPI drawing calls on page 22-9.) Each segment in the chain is called a *root segment.* Root segments are affected by those GPI calls that act on the segment chain, but they can also be manipulated independently of the chain.

Segments that are defined with the chained attribute switched off are said to be *unchained segments.* Unchained segments are always retained when they are created, regardless of the current drawing-mode parameter (if the drawing-mode parameter is DM_DRAWANDRETAIN or DM_DRAW, an unchained segment is retained but not drawn). An unchained segment must have a unique name.

There are a number of reasons for defining a segment as unchained. For example, a particular segment may not belong to the picture that is defined by the segment chain. You are also likely to define as unchained any segment that belongs to the picture, but that has no single, fixed place in the segment chain. A car wheel, for example, could be defined once but drawn four times in a drawing of a car. It would have no single, fixed place in the segment chain, but would be included four times in the picture by being *called* from one or more root segments.

A closed, unchained segment can be called from any other segment. Chained segments, however, cannot be called from other segments. A segment and the segments it calls are logically one object. A segment is called from another segment by including a GpiCallSegmentMatrix (GICALM) call in the calling segment. You can find out more about GpiCallSegmentMatrix in Chapter 25, "Transformation." An important point about called segments is that they assume the primitive-attribute settings of the calling segment. You can, of course, change the attribute settings within the called segment if the inherited values are inappropriate.

Figure 22-1 illustrates the segment chain concept.

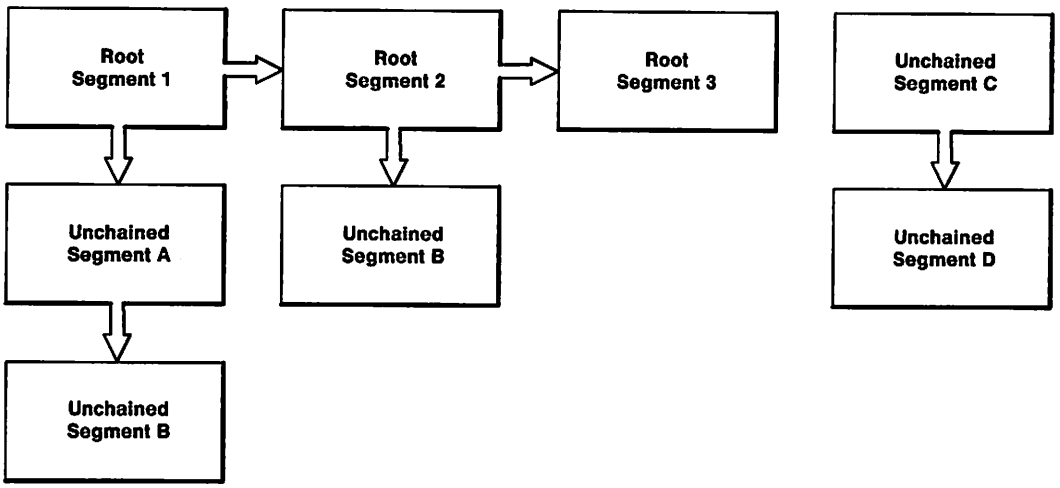


Figure 22-1. The Segment Chain. Segment A is called by root segment 1, and segment B is called by both segment A and root segment 2. Segment C calls segment D. Both are unchained segments that are not called from the segment chain, and are consequently not part of the segment chain.

When a segment chain is drawn, the segments are drawn in the order in which they appear on the chain. For example, the segments in Figure 22-1 are drawn in the following order:

1. Root segment 1
2. Unchained segment A
3. Unchained segment B
4. Root segment 2
5. Unchained segment B
6. Root segment 3.

The order of the segments, in conjunction with the current foreground and background mix modes, can affect the way in which the drawing appears to the user. The GpiSetSegmentPriority (GSSPRI) call changes the position of a root segment (which must not be a zero segment) in the chain. As input to this call, you supply the name of the segment you want to reorder, and the name of a reference segment. The reference segment is the segment that will be immediately before, or immediately after, the reordered segment when it is in its new position in the chain. If the segment you are moving is to come after the reference segment in the chain, it is said to have a higher priority (HIGHER_PRI). If it is to come before the reference segment, it is said to have a lower priority (LOWER_PRI). The nearer a segment is to the end of the chain, the higher its priority.

For example, to reposition root segment 3 from Figure 22-1 between segments 1 and 2, the reference segment can be root segment 1 (HIGHER_PRI) or root segment 2 (LOWER_PRI). The chain would be drawn in the following order:

1. Root segment 1
2. Unchained segment A
3. Unchained segment B
4. Root segment 3
5. Root segment 2
6. Unchained segment B.

If you supply the name of an unchained segment on input to GpiSetSegmentPriority (GSSPRI), the segment is added to the chain in the position you specify. To find out the position of a segment in the chain, you use the GpiQuerySegmentPriority (GSQPRI) call.

The Fast-Chaining Attribute

The fast-chaining attribute applies only to chained segments. It prevents primitive attributes from being reset to their default values at the beginning of the segment. This can be considered a performance aid, as there is an unnecessary overhead in resetting attributes to their defaults if:

- You are going to set attributes to nondefault values at the start of the segment, or
- You know that attributes have not previously been set to nondefault values.

The fast-chaining attribute is switched on by default, and you should leave it switched on unless you actively want attributes to be set to their default values. For more information about the fast-chaining attribute, see Appendix C, "Graphics Attributes"

The Dynamic Attribute

Dynamic segments are graphics segments that, after being drawn on the screen, can be moved to a different coordinate position (or altered in some other way) without affecting the rest of the picture. The new position of a dynamic segment can be provided by operator use of an input device, such as a mouse, or it can be application-generated.

The dynamic attribute is switched off by default. The setting is always inherited by called segments from the setting of their callers. Thus, if a calling segment is not dynamic, any segments called from it are also nondynamic, regardless of how they have been defined. If the calling segment is dynamic, all segments called from it are also dynamic. Any segment explicitly defined as dynamic must have a unique name, and it cannot be created when the current drawing-mode parameter is DM_DRAW or DM_DRAWANDRETAIN. Although no error condition is raised if you define an unchained segment as dynamic, it is not treated as dynamic unless it is called from a dynamic root segment.

As explained in the description of mix modes on page 21-6, graphics objects to be moved without disturbing the contents of the rest of the display are drawn in exclusive-OR mode. Segments that you define as dynamic are always drawn in exclusive-OR mode, regardless of the current mix-mode settings and of any GpiSetMix (GSMIX) calls the segment itself may contain.

For performance reasons, dynamic segments should be grouped together at the start of the picture chain. There are GPI calls that handle dynamic segments as a group, so it is more efficient if the whole segment chain does not have to be scanned to locate them. When the whole picture chain is drawn, however, dynamic segments should be drawn *after* all other segments in the chain to ensure that the effects of drawing in exclusive-OR mode are not compromised by drawings in other mix modes. Also, you must ensure that no nondynamic drawing cuts across the dynamic segments after they have been drawn. The Presentation Manager provides a control to make sure that dynamic segments are always drawn on top of other graphics automatically. This is described in "The Drawing Controls" on page 22-10.

The Detectability Attribute

The detectability attribute of a segment determines whether that segment can be selected by operator use of an input device, such as a mouse. A segment with this attribute is said to be *pickable*. Pickable segments are involved in *correlation operations*, which are described on page 22-12. The detectability attribute is switched off by default. All detectable segments must have unique names. If you define a zero segment as detectable, it is created as nondetectable.

The Propagate-Detectability Attribute

This attribute controls whether the detectability attribute of a calling segment is inherited by any segments called from it. The propagate-detectability attribute is switched on by default. This value overrides the detectability setting of the called segment. For example, if you switch off the detectability attribute of a calling segment, any segments called from it are nondetectable, regardless of how they have been defined.

The Visibility Attribute

The visibility attribute controls whether a segment is to be visible on the output device. The contents of a segment that is not defined with the visibility attribute switched on are still executed whenever the segment is drawn. This can cause changes to the current position and to current attribute values, even though the primitives themselves are not visible on the output device. The visibility attribute is switched on by default.

The Propagate-Visibility Attribute

This attribute controls whether the visibility attribute of a calling segment is inherited by any segments called from it. The propagate-visibility attribute is switched on by default. This value overrides the visibility setting of the called segment. For example, if you switch off the visibility attribute of a calling segment, any segments called from it are not visible, regardless of how they have been defined.

Changing the Attributes of a Segment

You can change the default segment-attribute settings globally (that is, for a single presentation space) using the GpiSetInitialSegmentAttrs (GSSATI) call. The attribute setting that you specify on this call applies to all segments created subsequently in that presentation space, except that a nonretained segment can never have the dynamic attribute. For example, if you want all segments to be detectable, you can issue this call to change the setting for all of them *before* you create them. It cannot be used to change the attributes of existing segments.

You can also change the attributes of any single retained segment (not just those created subsequently) using the GpiSetSegmentAttrs (GSSATS) call. This is useful if, for example, you want most of the segments in a picture chain to be detectable. You can change the setting to detectable for the whole chain (before you create it) using GpiSetInitialSegmentAttrs (GSSATI), and use GpiSetSegmentAttrs (GSSATS) to change it back to nondetectable for each of the exceptions. You can also use GpiSetSegmentAttrs to make a visible segment invisible, for example, when an erase request is received from the operator, or to take a segment out of the chain by redefining it as unchained. If you make an unchained segment chained, it is added to the end of the segment chain. If you want it to be positioned elsewhere in the chain, use the GpiSetSegmentPriority (GSSPRI) call rather than GpiSetSegmentAttrs (GSSATS).

Some of these segment attributes apply equally to primitives outside segments, although their default settings cannot be changed. Primitives outside segments are always detectable and visible. They cannot be dynamic, because the dynamic attribute applies only to retained graphics. The chained and fast-chaining-attributes do not apply to primitives outside segments.

Drawing a Picture

Nonretained graphics segments and primitives outside graphics segments are always drawn immediately to the current output device. Segments that have been retained in segment store, however, can be drawn any number of times to any number of device contexts. There are GPI drawing calls that allow you to draw a single segment, a part of the segment chain, or the complete segment chain.

The GpiDrawSegment (GSDSEG) call draws a single named segment, which can be chained or unchained. You supply the segment name as input to this call. GpiDrawSegment can draw a dynamic segment in some circumstances. These are described in "Drawing Dynamic Segments" on page 22-10.

The GpiDrawFrom (GSDF) call draws one or more root segments from the segment chain. You supply the names of two root segments as input to this call. The drawing process starts at the first named segment and draws all chained and called segments, excluding dynamic segments, up to and including the second named segment. The order in which you specify the segment names should reflect their relative positions in the segment chain. Thus, if the second-named segment appears in the segment chain before the first-named segment, no error is raised, but drawing starts from the first named segment and continues to the end of the chain. This also happens if the second segment is an unchained segment rather than a root segment. If the drawing control DCTL_DYNAMIC is set, any dynamic segments already drawn on the display are removed before GpiDrawFrom (GSDF), and redrawn when it completes.

The GpiDrawChain (GSDCHN) call draws all nondynamic chained segments in a named presentation space, plus any unchained segments called by those chained segments. It does not draw dynamic segments. You supply only the presentation space handle as input to this call. If the drawing control DCTL_DYNAMIC is set, any dynamic segments already drawn on the display are removed before GpiDrawChain (GSDCHN), and redrawn when it completes.

You can issue any of these calls while there is an open segment. The open segment is not modified by these calls, though any of these calls can result in the current attribute values and current position being modified. For more information about the attribute-resetting process, see Appendix C, "Graphics Attributes." The current drawing mode has no effect on the GPI drawing calls.

If an error occurs during the drawing process, you can find out where the error occurred using the GpiErrorSegmentData (GSESD) call. This call returns information about the last error that occurred during a segment-drawing operation,

including the name of the segment, and exactly where in the segment the error occurred.

Drawing Dynamic Segments

Dynamic segments are segments that can be moved or altered in some other way without disturbing the rest of the picture. To draw dynamic segments on the screen for the first time, you use the GpiDrawDynamics (GSDDYN) call. GpiDrawDynamics draws every dynamic segment in the segment chain. When they have been drawn on the display, dynamic segments can be updated. To move one or more dynamic segments to another part of the display, for example, you remove them from their current position using the GpiRemoveDynamics (GSRDYN) call. You can restrict the scope of this call by supplying the names of a start segment and a finish segment. The start segment and the finish segment can be the same segment. Dynamic segments outside this range are not removed from the display. If you intend to re-associate the presentation space, you should issue the GpiRemoveDynamics (GSRDYN) call first. If you fail to do this, the GpiRemoveDynamics call cannot remove the dynamic segments after the presentation space has been re-associated. The dynamic segments effectively cease to be dynamic.

After removing the appropriate dynamic segments from the display, you can redraw them elsewhere on the display, using the GpiDrawDynamics (GSDDYN) call again. If you specify a range of dynamic segments on the GpiRemoveDynamics (GSRDYN) call, the redrawing is restricted automatically to that range. Note that, when dynamic segments have been drawn on the display, you must not edit their definitions in segment store.

Dynamic segments are treated like nondynamic segments when you are directing output to a hard-copy device. GpiRemoveDynamics (GSRDYN) and GpiDrawDynamics (GSDDYN) cause an error condition to be raised if the current output device is not a screen window.

The GpiDrawSegment (GSDSEG) call can be used to draw a dynamic segment in some circumstances. Its effects are as follows:

- If the named segment is both chained and dynamic, and the DCTL_DYNAMIC drawing control is set, *all* dynamic segments in the chain, including the named segment, are drawn as dynamic. (The drawing controls are described on this page.)
- If the named segment is both chained and dynamic, and the DCTL_DYNAMIC control is not set, the named segment is not drawn.
- If the named segment is unchained and dynamic, it is drawn as a nondynamic segment, regardless of the setting of the DCTL_DYNAMIC control.

The Drawing Controls

The GpiErase (GSERSE) call clears the output display of the currently associated device to CLR_BACKGROUND, which is the normal background color for the device. If you are using the default color table, this value clears the window background to white. This is a once-only request, and has to be issued every time you want the display cleared before drawing a new picture. You can use this call for a micro presentation space or a normal presentation space.

To get this effect for more than one drawing request, you can use the GpiSetDrawControl (GSDC) call. This call establishes current values for five drawing controls, including the erase-before-draw control (DCTL_ERASE) which remain in effect until they are reset. The display control (DCTL_DISPLAY) for example, can be switched off to prevent drawings appearing on an output device, and the draw-dynamic-segments control (DCTL_DYNAMIC) automatically removes all dynamic segments from the screen before any GpiDraw request, and redraws them again afterward. This ensures that dynamic segments are always drawn on top of nondynamic segments and primitives. The correlation control (DCTL_CORRELATE) is described in "Correlating on Nonretained Graphics" on page 22-14. The boundary-data-accumulation control (DCTL_BOUNDARY) is described in Chapter 26, "Clipping."

GpiSetDrawControl can be issued to a micro or a normal presentation space, though not all of the controls are valid in a micro presentation space. For details, see the *Programming Reference: Volume 1*.

Attribute Modes

You can change primitive-attribute settings at any time, and the new values you specify replace the existing values. If the current foreground color is green and you change it to red, for example, red replaces green.

When you set primitive attributes within a retained or nonretained segment, you can save the existing attribute values on a LIFO (last in, first out) stack, from where they can later be retrieved and made current again. You do this by selecting an *attribute mode*.

There are two attribute modes, which are AM_PRESERVE mode (also known as *push-and-set mode*) and AM_NOPRESERVE mode (also known as *set mode*). In AM_NOPRESERVE mode, which is the default setting, existing attribute values are replaced by any new attribute values that you supply. In AM_PRESERVE mode, the existing attribute values are stored on a LIFO stack, and any new values that you specify take effect.[1] Those attributes that are affected by the current attribute mode are listed in the *Programming Reference: Volume 1*.

To select a current attribute mode, you use the GpiSetAttrMode (GSAM) call. The current attribute mode affects subsequent attribute-setting requests in the presentation space. Attribute modes are not applicable to micro presentation spaces, because graphics segments can be created only in a normal presentation space.

To retrieve an attribute value from the LIFO stack, you use the GpiPop (GSPOP) call. This call pulls back the attribute that was most recently stored on the stack. You can get back more than one attribute value by supplying a number as an input parameter of this call. For example, if you specify '4', the GpiPop (GSPOP) call retrieves the four attributes that were most recently stored on the stack. If each of the values retrieved applies the same type of attribute to the same primitive (for example, if all four are line-type settings), the last one to be retrieved (and therefore the first one on to the stack) becomes the current setting.

If you save attribute values from any segment called from another segment, and do not retrieve the values using GpiPop (GSPOP), they are restored automatically when the end of the segment is reached. If you save attribute values from any segment that is not called from another segment, and do not explicitly restore those values using GpiPop, they are lost at the end of the segment.

AM_PRESERVE mode is useful when you do not want calling segments to inherit attribute settings from the segments they call. (This happens because a calling segment and the segments it calls are logically one object, so attribute changes within a called segment remain current on return to the calling segment.) If you set some attribute values at the start of a called segment when the current attribute mode is AM_PRESERVE, the attribute values of the calling segment are stored on the LIFO stack. At the end of the called segment, the values on the stack are automatically retrieved, so that the calling segment continues with its own attribute values.

Reusing the Presentation Space

A normal presentation space with segments retained in segment store can be expensive in storage terms. You should therefore delete any presentation space that you no longer need by issuing the GpiDestroyPS (GIPDEL) call. If your application is using a series of presentation spaces, each with a different format, for example, the creation and deletion activity can also be expensive. The Presentation Manager provides a number of calls that let you reuse or redefine an existing presentation space, so you can create a presentation space once, and use it any number of times.

The calls are:

- GpiSavePS
- GpiRestorePS
- GpiResetPS
- GpiSetPS.

[1] AM_PRESERVE mode also causes the current position to be saved if the position was specified using the GpiSetCurrentPosition (GICP) call. If you use the GpiMove (GIMOVE) call to set the current position, the position is not saved, regardless of the attribute mode.

The GpiSavePS (GSSPS) Call

The GpiSavePS (GSSPS) call saves presentation-space information (such as current primitive attributes and the current position) on a dedicated LIFO (last in, first out) stack. The presentation space itself is unchanged; that is, it still exists, it has the same presentation-space handle, and the presentation-page dimensions and format are the same.

Output from GpiSavePS (GSSPS) is a number that identifies the saved information on the LIFO stack. An output of '3', for example, tells you that this is the third lot of presentation-space data on the stack.

The GpiRestorePS (GSRSPS) Call

To retrieve the saved information from the LIFO stack and reapply it to the presentation space using the GpiRestorePS (GSRSPS) call. Input to this call can be either the identifier returned to you from the GpiSavePS call, or a relative value. A value of −1, for example, retrieves the information most recently stored on the stack. You do not have to restore the information that was most recently saved on the stack. However, any data that you skip over is discarded. For example, if there are four lots of data on the stack and you request the restoration of lot 2 (which you do by specifying either an absolute value of '2', or a relative value of '−3'), lots 3 and 4 are discarded, and lot 1 is the only one remaining on the stack. The contents of the LIFO stack are purged whenever GpiAssociate (GSASS) is issued.

The GpiResetPS (GSRPS) Call

When a graphics presentation space is first created, it is in a neutral state. For example, current attributes are set to their default values, the current position is (0,0), the segment store contains no segments, and there are no application-defined resources, such as logical color tables. You can return an existing presentation space to this state using the GpiResetPS (GSRPS) call. GpiResetPS has three levels of reset-activity, each more powerful than the last. These are:

GRES_ATTRS, which is equivalent to the processing caused by a GpiCloseSegment (GSSCLS) call.

GRES_SEGMENTS, which is equivalent to creating a new presentation space, but without deleting any logical resources. All retained segments are deleted from segment store.

GRES_ALL, which is equivalent to creating a new presentation space.

The GpiResetPS (GSRPS) call does not alter the size or the format of the presentation page.

The GpiSetPS (GIPS) Call

To redefine the size and the format of the presentation page, you use the GpiSetPS (GIPS) call. The processing performed when you issue GpiSetPS is similar to that performed by GpiCreatePS (GIPCRT), except that the presentation space already exists. That is, the presentation space is returned to a neutral state (which is the equivalent of requesting GRES_ALL on the GpiResetPS call), and the presentation page is redefined. For example, you can change a presentation page defined in 0.01 mm units to one defined in pels. This lets you work with a 'new' presentation space without having to delete one and create another.

You can also use the GpiSetPS to change the current mapping mode of a presentation space. To do this, you use the PS_NORESET flag, which is the equivalent of requesting GRES_SEGMENTS on the GpiResetPS call. This feature is particularly useful if you are designing an application that deals with page layout and drafting, or if you want the screen size to correspond to the page size for the printed output.

Correlation

When you want to pick an item on the screen, you usually position the pointer at an appropriate point and signal (by clicking a mouse, for example) that this is the item you want. This action communicates an (x,y) coordinate position to the application. However, the application has no way of knowing exactly what you are trying to select. There might be overlapping segments at that coordinate position, or there might be a number of segments equidistant from it. The application handles this by defining a small rectangle centered on the (x,y) position you select, and calculating which detectable segments, if any, intersect that rectangle.

This process is known as *correlation*, and the notional rectangle that the application generates is known as the *pick aperture*. The pick aperture is shown in Figure 22-2.

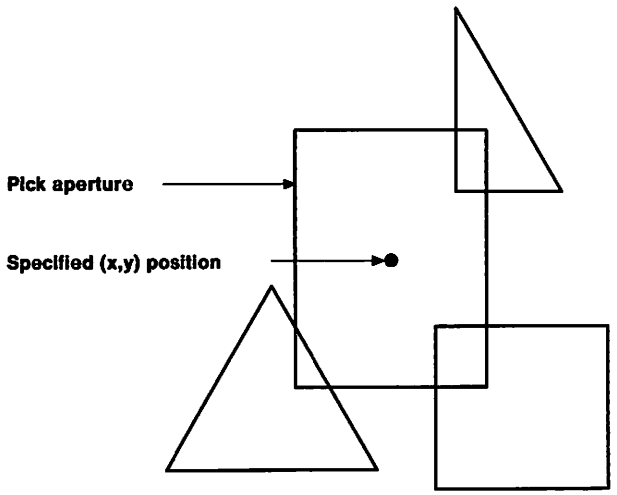


Figure 22-2. The Pick Aperture. Three separate segments intersect this pick aperture. However, the box shape is the only one that can be picked, because it is the only one that is defined as detectable.

The Pick Aperture

You specify the size of the pick aperture using the GpiSetPickApertureSize (GISPAS) call. You should aim to retrieve a single detectable object for each correlation operation, so you have to obtain a balance between the number of detectable objects in a picture, and the size of the pick aperture. The more detectable objects you define, the smaller the pick aperture needs to be to return a manageable amount of information to the application.[2]

Correlating on Retained Graphics

For any retained segment to be a candidate for correlation:

- It must have a unique identifier,
- It must be nondynamic, and
- It must be defined as detectable.

For an operator to be able to pick a segment explicitly, the segment should also be visible, although an application can still receive correlation data about invisible segments that intersect the pick aperture.

Each primitive or group of primitives within a detectable segment must be labeled with a *pick tag*. To specify a pick tag, you use the GpiSetTag (GSTAG) call. The pick tag is usually greater than zero. A pick tag of zero has the effect of turning *off* detectability for subsequent primitives. You can use this to make parts of a segment detectable, for example, and parts of it nondetectable. The pick tag is zero by default.

The tag you specify becomes the current tag, and it applies to all subsequent primitives until you next issue the GpiSetTag (GSTAG) call. The tag can be considered one of the attributes of the primitive, and is, therefore, affected by the current attribute mode. In AM_PRESERVE mode, it is stored on the LIFO stack, and can be recalled by the GpiPop (GSPOP) call.

The Correlation Calls

The Presentation Manager lets you correlate on a segment chain (GpiCorrelateChain (GICORS)), on a range of segments from a segment chain (GpiCorrelateFrom (GICORF)), and on a single segment (GpiCorrelateSegment (GICSEG)). As input to these calls you supply the identifiers of the chain or segments, and the coordinate position on which the pick aperture is to be centered. This has usually been obtained from operator input, as shown in this example:

```
GpiDrawChain
  /* Operator input is received, such       */
  /* as WM_BUTTON1DOWN message              */
GpiSetPickApertureSize
  /* Specify the size of the pick           */
  /* aperture.                              */
GpiCorrelateChain
  /* Position the pick aperture using the   */
  /* operator-supplied position, and return */
  /* correlation data to the application.   */
```

[2] If the pick aperture is wholly contained within a box or full arc primitive, no correlation data is returned to the application unless the primitive has been defined as filled.

You can request correlation data:

- For those segments that have been defined as both detectable and visible, or
- For all nonzero segments that intersect the pick aperture, regardless of their detectability and visibility attributes.

The names of segments within the pick aperture are returned to the application in reverse order of their occurrence on the segment chain. For example, if the first, second, and third segments in a chain intersect the pick aperture, they are returned in the order 3,2,1. That is, the highest priority segment is returned first. The application can therefore identify the topmost detectable and visible segment, which is the segment most likely to have been picked.

The data returned from each correlation operation consists of a set of segment and tag pairs. Each unique segment and tag pair within the pick aperture is known as a *hit*. If two or more primitives in the pick aperture have the same tag, they are considered a single hit. For example, if Segment 6 contains two groups of four primitives, and the groups have the tags 4 and 5, correlation on that segment returns the two hits:

```
6 - 4
6 - 5
```

if at least one primitive from each group intersects the aperture. As correlation hits are returned in reverse order, these two hits would be returned in the order 6 − 5, 6 − 4. If only one hit is requested, the hit 6 − 5 would be returned. If the primitives in segment 6 are not tagged, no correlation hits are returned to the application.

When a called segment is picked, correlation data is also returned for all segments above it in the hierarchy, up to and including the root segment, and this still constitutes a single hit. You can limit the number of segment and tag pairs returned for each hit using the maxdepth parameter, and you can limit the total number of hits returned to you using the maxhits parameter.

Correlating on Nonretained Graphics

The need to correlate on graphics can usually be predicted, and therefore can be planned for when you are designing applications. This does not mean, however, that you have to create retained rather than nonretained segments. The decision to retain graphics is dependent on several considerations, of which correlation is only one.

Nonretained graphics are, for the purposes of correlation, those graphics that are being correlated on *during* the drawing process. In this context, a nonretained graphic is one created in draw mode, or, if created in draw-and-retain mode, one that has not yet been retained. If they are to be correlated on, nonretained segments must have unique identifiers, and they must be defined as detectable. The primitives in the segment can be tagged, though this is not required. Primitives outside segments are automatically detectable.

To get correlation data from the drawing of nonretained graphics, you have to switch on the *correlate flag* (DCTL_CORRELATE) of the GpiSetDrawControl (GSDC) call first. If the size of the pick aperture is not suitable, you can change it using the GpiSetPickApertureSize (GISPAS) call. You must then position the aperture explicitly using the GpiSetPickAperturePosition (GIPAP) call. As input to this call you provide the coordinate position on which the pick aperture is to be centered.

As individual primitive-drawing requests, GpiPutData calls (GSPUT), GpiElement calls, and GpiPlayMetaFile (GSLOAD) calls are executed, correlation is performed to determine which graphics objects intersect the pick aperture. You detect correlation hits by examining the returned values from the GPI calls. For example, if a GpiLine (GILINE) request causes a line to intersect the pick aperture, there is a return value of GPI_HITS from GpiLine to indicate a correlation hit.

Example Code: Creating Segments

This example, which shows how to create chained and unchained segments, is taken from the Graphic1 sample program. Graphic1 draws 2 box shapes, and several small irregular shapes, colored blue or red, that can be moved into the 2 boxes. In addition to showing most of the concepts described in this chapter, Graphic1 introduces transformations. Transformations are described in Chapter 25, "Transformation."

```
VOID StorePicture()
{
  static LONG lBoxWidth = 40;
  static LONG lBoxHeight = 40;
  static LONG alNumpts[LASTPART-FIRSTPART+1] = {4,3,3,6,4,3,4,5,4};
  static POINTL aptPart[LASTPART-FIRSTPART+1][6]
                              = {10,00, 10,20, 00,30, 0,0,   0,0,  0,0,
                                 00,10,-10,10, 0,0,   0,0,   0,0,  0,0,
                                 00,10,-10,10, 0,0,   0,0,   0,0,  0,0,
                                 00,20, 20,20, 30,10, 10,10, 10,0, 0,0,
                                 10,00, 10,10,-10,10, 0,0,   0,0,  0,0,
                                 20,20, 00,20, 0,0,   0,0,   0,0,  0,0,
                                 10,00, 10,10, 00,10, 0,0,   0,0,  0,0,
                                 20,00, 20,20, 10,20, 10,10, 0,0,  0,0,
                                 30,00, 20,10, 10,10, 0,0,   0,0,  0,0};

  LONG   lIwork, lJwork;
  POINTL ptlWork;
  SIZEF  sizfCharBox;
  LONG   lSaveMode;


  GpiSetDrawingMode(hpsClient, DM_DRAWANDRETAIN);
  GpiSetInitialSegmentAttrs(hpsClient, ATTR_CHAINED, DCTL_OFF);
  GpiSetInitialSegmentAttrs(hpsClient, ATTR_FASTCHAIN, DCTL_OFF);

  /*********************************************************************/
  /* The drawing mode is set, and the chained and fast-chaining        */
  /* segment attributes are turned off. The next section of code       */
  /* creates several unchained, retained segments. For each segment, a */
  /* different polyline definition is taken from the arrays alNumpts   */
  /* and aptPart.                                                      */
  /*********************************************************************/

  for (lIwork=FIRSTPART ; lIwork<=LASTPART ; lIwork++){
    GpiOpenSegment(hpsClient, lIwork);
    GpiSetTag(hpsClient, lIwork);
    ptlWork.x = 0;
    ptlWork.y = 0;
    lSaveMode = GpiQueryAttrMode(hpsClient);
    GpiSetAttrMode(hpsClient, AM_PRESERVE);
    GpiSetAttrs(hpsClient,
                PRIM_LINE,
                LBB_COLOR,
                0L,
               (PBUNDLE)&LineAttr
               );

  /*********************************************************************/
```

```
/* The current attribute mode is saved before being set to          */
/* AM_PRESERVE. The line-color attribute is set.                    */
/********************************************************************/

  GpiBeginPath(hpsClient, (LONG)1);
  GpiSetCurrentPosition(hpsClient, &ptlWork);
  GpiPolyLine(hpsClient,                    /* draw predefined shape */
              alNumpts[lIwork-FIRSTPART],
              aptPart[lIwork-FIRSTPART]);
  GpiCloseFigure(hpsClient);
  GpiEndPath(hpsClient);

/********************************************************************/
/* A path comprising one figure is defined.                         */
/********************************************************************/

  GpiLabel(hpsClient, FILLPATH);

  GpiFillPath(hpsClient, (LONG)1, FPATH_ALTERNATE);
  GpiSetCurrentPosition(hpsClient, &ptlWork);
  GpiPolyLine(hpsClient,
              alNumpts[lIwork-FIRSTPART],
              aptPart[lIwork-FIRSTPART]);
  GpiSetAttrMode(hpsClient, lSaveMode);
  GpiPop(hpsClient, (LONG)1);

  GpiCloseSegment(hpsClient);
} /* endfor */

/********************************************************************/
/* The label FILLPATH is added to the segment to make editing easier,*/
/* and the path is filled in the current area-fill pattern. The     */
/* shape of the path is drawn a second time to give the path a      */
/* contrasting border. The attribute mode is restored, and the      */
/* attribute values are retrieved from the LIFO stack.              */
/********************************************************************/

GpiSetInitialSegmentAttrs(hpsClient, ATTR_CHAINED, DCTL_ON);
lJwork = 1;

/********************************************************************/
/* The chaining attribute is switched on. Each of the next two      */
/* segments draws a box. The first box is filled with the color     */
/* green, and contains the character string 'RED'. The second box is */
/* filled with the color yellow, and contains the character string  */
/* 'BLUE'. Both character strings are centered.                     */
/********************************************************************/

GpiOpenSegment(hpsClient, lJwork++);
GpiSetColor(hpsClient, CLR_GREEN);
ptlWork.x = 0;
ptlWork.y = 0;
GpiSetCurrentPosition(hpsClient, &ptlWork);
ptlWork.x += lBoxWidth;
ptlWork.y += lBoxHeight;
GpiBox(hpsClient, DRO_OUTLINEFILL, &ptlWork, (ULONG)NULL, (ULONG)NULL);
GpiSetColor(hpsClient, CLR_BLACK);

sizfCharBox.cx = 8L<<16;
sizfCharBox.cy = 20L<<16;
```

```
GpiSetCharBox( hpsClient, &sizfCharBox );
GpiSetCharMode(hpsClient, CM_MODE3);

ptlWork.x = (lBoxWidth - LOUSHORT(GetTextSize(hpsClient, "RED")))/2;
ptlWork.y = (lBoxHeight - HIUSHORT(GetTextSize(hpsClient, "RED")))/2;
GpiCharStringAt(hpsClient, &ptlWork, (ULONG)3, (PSZ)"RED");
GpiCloseSegment(hpsClient);


GpiOpenSegment(hpsClient, lJwork++);
GpiSetColor(hpsClient, CLR_YELLOW);
ptlWork.x = 0;
ptlWork.y = 80;
GpiSetCurrentPosition(hpsClient, &ptlWork);
ptlWork.x += lBoxWidth;
ptlWork.y += lBoxHeight;
GpiBox(hpsClient, DRO_OUTLINEFILL, &ptlWork, (ULONG)NULL, (ULONG)NULL);
GpiSetColor(hpsClient, CLR_BLACK);

sizfCharBox.cx = 8L<<16;
sizfCharBox.cy = 20L<<16;
GpiSetCharBox( hpsClient, &sizfCharBox );
GpiSetCharMode(hpsClient, CM_MODE3);

ptlWork.x = (lBoxWidth - LOUSHORT(GetTextSize(hpsClient, "BLUE")))/2;
ptlWork.y = 80 + (lBoxHeight - HIUSHORT(GetTextSize(hpsClient, "BLUE")))/2;
GpiCharStringAt(hpsClient, &ptlWork, (ULONG)4, (PSZ)"BLUE");
GpiCloseSegment(hpsClient);

GpiSetInitialSegmentAttrs(hpsClient, ATTR_DETECTABLE, DCTL_ON);

/*********************************************************************/
/* The next part of the segment chain is both chained and detectable.*/
/* The function of this part of the chain is to draw the unchained   */
/* segments, first in red and then in blue. Each time a segment is   */
/* drawn, an instance transformation is specified to position the    */
/* segment.                                                          */
/*********************************************************************/

for (lIwork=FIRSTPART ; lIwork<=LASTPART ; lIwork++){
  GpiOpenSegment(hpsClient, lJwork);
  GpiSetTag(hpsClient, lJwork);
  GpiSetColor(hpsClient, CLR_RED);
  GpiLabel(hpsClient, SEGCALL);
  GpiCallSegmentMatrix(hpsClient,
                       lIwork,
                      (ULONG)9,
                      &amatlfSegXform[lJwork++],
                       TRANSFORM_ADD);
  GpiCloseSegment(hpsClient);
} /* endfor */

for (lIwork=FIRSTPART ; lIwork<=LASTPART ; lIwork++){
  GpiOpenSegment(hpsClient, lJwork);
  GpiSetTag(hpsClient, lJwork);
  GpiSetColor(hpsClient, CLR_BLUE);
  GpiLabel(hpsClient, SEGCALL);
  GpiCallSegmentMatrix(hpsClient,
                       lIwork,
                      (ULONG)9,
```

```
                                &amatlfSegXform[lJwork++],
                                 TRANSFORM_ADD);
      GpiCloseSegment(hpsClient);
    } /* endfor */
```

Chapter 23. Graphics Text

You can produce alphanumeric text using the Presentation Manager VIO functions, which are described in Chapter 30, "Alphanumeric Video Output." VIO text is useful for simple monospaced text of limited quality. If you require high-quality text, or if you want to mix text and graphics, you use the graphics programming interface to produce graphics characters. All graphics characters are defined in *fonts*. A font is a set of character definitions in a particular size and style. The characters in the font can be defined either as *images* or as *outlines*.

Each of the characters in an *image font* (sometimes called a raster font) is described in a rectangular array of pels. Some of the pels in the array are set to produce the image of a character. Image fonts are usually of good quality, though the characters have to be drawn in a fixed size. You cannot scale or rotate image characters, for example.

Each of the characters in an *outline font* (sometimes called a vector font) is drawn using a sequence of lines and arcs, and is then filled. Outline characters can be slow to print or display because of the way they are drawn. The advantage of outline fonts is that the characters can be scaled, rotated, and sheared. Outline fonts are therefore said to be *transformable*. Figure 23-1 shows an example of a character from an image font and an outline font.

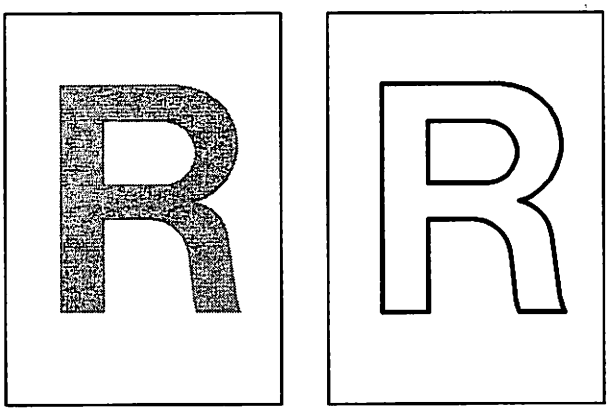


Figure 23-1. Image and Outline Characters. The image character is produced by setting some of the pels in a rectangular array. The outline character is produced by drawing a sequence of lines and arcs, and then filling the resultant shape.

The Presentation Manager-Supplied Fonts

Presentation Manager provides a number of typographic-quality character fonts. They are listed in Table 23-1. These fonts are available on all OS/2-supported display adapters and printers.

In addition, there is a default system font that is used in window components, such as title bars and menus. It is a proportionally-spaced Swiss font.

Table 23-1. The Presentation Manager Character Fonts

Point Size	Roman	Swiss	Courier
8-point	√	√	√
10-point	√	√	√
12-point	√	√	√
14-point	√	√	X
18-point	√	√	X
24-point	√	√	X

The Courier font is not available in 14- , 18- , and 24-point sizes.

Type size is measured in *points*. There are approximately 72 points to an inch, so each character in a 24-point font, for example, is 1/3-inch high. Each incremental unit of a presentation page in PU_TWIPS format is a twentieth of a printer's point (1/1440 in.).

The fonts listed in Table 23-1 are image fonts. In addition, each font has an outline version in normal, bold, italic, and bold italic. On some devices, image fonts (in the small point sizes) generally have a better appearance than their outline-font equivalents.

Creating Fonts

You cannot edit the definitions of any of the Presentation Manager-provided fonts, although you can create and edit image font-files using the Presentation Manager-supplied Font Editor. A font file contains a header, which describes the font in

general terms, and a section containing bit maps of the characters themselves. The font definition is stored in a *resource file*. For information about the Font Editor and about resource files, see "The Font Editor" on page 8-8. A full description of the font-file format is provided in the *Programming Reference: Volume 2*.

Public and Private Fonts

The Presentation Manager-supplied fonts are loaded automatically when the system is started, and are available to any application at any time. They are said to be *public fonts*. Any font that you define locally has to be loaded before you can use it. If you load a locally-defined font using the Install option of the Control Panel, that font is also available to any application in the system, and is therefore a public font.

Any font loaded using the GpiLoadFonts (GSFLO) call is said to be a *private font*. A private font is available only to the application that loads it. The file in which you store a locally created font definition can contain more than one font definition. GpiLoadFonts (GSFLO) loads a named dynamic-link-library file, and therefore loads all fonts defined in that file. The GpiQueryFontFileDescriptions (GSQFD) call tells you the family name and the facename of each font in a locally defined font file. You use this call to find out whether a particular file contains the font you want to use before you load it. To retrieve the names of all font files available on the system, you use the WinQueryProfileString (WMQPST) call and specify the application name PM_FONTS.

To unload a font file, you use the GpiUnloadFonts (GSFUL) call. You cannot unload public fonts using GpiUnloadFonts.

You cannot use private fonts in a document that you want to print using the spooler if you create a PM_Q_STD-format spool file. However, you can use either public or private fonts if you specify a PM_Q_RAW-format spool file, or if you want to print without the spooler. If you want to print the document from a printer server, all the fonts used in the document must be installed as public fonts on the server. For more information on spool-file formats, see Chapter 36, "Printing."

Selecting a Font

Every presentation space has a *current font*, and graphics characters are always drawn from the current font. By default, the current font is the system font. Before an application can use any font other than the system font, it has to identify that font as the current font. There are two steps in this process:

1. The first step is to define a *logical font* using the GpiCreateLogFont (GSCRLF) call. (The Presentation Manager-supplied fonts, and any that you load using the Control Panel or the GpiLoadFonts (GSFLO) call, are *physical fonts*.) GpiCreateLogFont (GSCRLF) establishes a link between the issuing application and an appropriate physical font. An application can have up to 254 logical fonts defined at any one time in a single presentation space.
2. The second step is to issue the GpiSetCharSet (GSCS) call to identify a logical font as the current font.

Defining a Logical Font

There are two possible approaches to defining a logical font:

1. You can use the GpiQueryFonts (GSQF) call to request information about the available physical fonts, and then explicitly select one of those fonts by supplying its identifier on input to GpiCreateLogFont (GSCRLF), or
2. You can use GpiCreateLogFont (GSCRLF) to specify the features you require in a font, and allow the Presentation Manager to make an appropriate font available to you.

GpiCreateLogFont (GSCRLF) has a number of input values, the most important of which are the *font attributes*. The font attributes describe the physical features and capabilities of the font. Whether you are naming a specific font, or describing the font you require, you should complete the entries in the FATTRS structure. This ensures that, if you are selecting a particular font and that font is unavailable, an attempt is made to find the most suitable substitute. Without the FATTRS information, the Presentation Manager is less likely to locate a suitable font. If you have issued GpiQueryFonts (GSQF), you can simply copy the font metrics of the font you want into the

FATTRS structure. The font attributes are described below.

The Font Attributes

FATTRS contains the following values:

Length
Length of the FATTRS structure.

Selection indicators
There are five selection indicators. These are:

FATTR_SEL_ITALIC, which indicates that you require a font with italic capabilities.

FATTR_SEL_BOLD, which indicates that you require a bold version of the font.

FATTR_SEL_OUTLINE, which indicates that you require hollow characters when using an outline font. This offers improved performance and may be acceptable when creating a draft or using a small character size.

FATTR_SEL_STRIKEOUT, which indicates that you require a font in which the characters are struck through.

FATTR_SEL_UNDERSCORE, which indicates that you require a font in which the characters are underscored.

You can set any combination of these indicators. When set, they cause *simulation* of that feature. For example, if you require a bold, italicized version of a Presentation Manager image font, the Presentation Manager simulates that feature because bold, italic image fonts are not supplied.

Match value
The match value is a number that identifies a physical font, and is allocated to the font when it is loaded. If the match value is <0, the font is a *device font*, and if the match value is >0, the font is a *generic font*. A device font is exclusive to the device with which it is associated, and its match number is guaranteed unique only within the device context. Printer fonts are usually device fonts. A generic font is available on more than one device. The Presentation Manager-supplied fonts, for example, are generic fonts.

If you are not selecting a specific font, you set the match value to 0.

Facename
The facename is the typeface name by which a particular font design is known. Two examples are Tms Rmn Bold and Tms Rmn Bold Italic. Roman is the *family name* of these fonts. You are recommended always to provide a facename.

Registry identifier
The registry number of a particular font can be retrieved by issuing the GpiQueryFonts (GSQF) call. If you do not have the registry number, set this value to 0.

Code page
This is the identifier of the code page to be associated with the font. For more information, see "The Code Page" on page 23-6.

Maximum baseline extent
The maximum baseline extent is the vertical space occupied by a character in the font. In conjunction with the font's external leading, the maximum baseline extent defines the line-spacing you require. If you are setting the font-use indicator FATTR_FONTUSE_OUTLINE, you can set the maximum baseline extent to 0. (Outline fonts take an equivalent value from the character-box attribute that is current when text is being written to an output device.) The maximum baseline extent value is required for an image font, and must be specified in world coordinates.

The maximum baseline extent measurement is shown in Figure 23-16 on page 23-16.

Average character width
The average character width of the font is the average character increment in world coordinates. The character increment is the sum of the a-space, the b-space, and the c-space, which are shown in Figure 23-2 on page 23-7. If you are setting the font-use indicator FATTR_FONTUSE_OUTLINE, you can set the average character width to 0. (Outline fonts take an equivalent value from the character-box attribute that is current when text is being written to an output device.) The average-character-width value is required for an image font, and must be specified in world coordinates.

Type indicator

The type indicator FATTR_TYPE_KERNING causes kerning to take effect for those fonts that contain kerning-pair information. Kerning is a device-dependent function. Kerning is described on page 23-7. This value is not interchangeable with the type indicators in the FONTMETRICS structure.

Font-use indicators

The font-use indicators tell the Presentation Manager what you intend to do with the font, and they determine whether you get an image font or an outline font.

To force selection of an outline font, you set on the FATTR_FONTUSE_OUTLINE indicator. This value must be set if you intend to use graphics characters in a path definition.

The FATTR_FONTUSE_TRANSFORMABLE indicator must be set if you want to scale, rotate, or shear characters from the font. It must also be set if you intend to use the font with a current character mode of CM_MODE3. (Character modes are described on page 23-8.)

If you set on the FATTR_FONTUSE_NOMIX indicator, the Presentation Manager can ignore current mix-mode values. You should set this value if the graphics text is not going to be written on top of, or underneath, other graphics, as it improves the performance of text drawing.

The two ways of defining a logical font are described below.

Specifying a Nonzero Match Value

To be able to select a particular font, you need to get information about the available physical fonts. To do this, you use the GpiQueryFonts (GSQF) call. GpiQueryFonts returns the *font metrics* of every font that is available to you. The font metrics define in detail the physical characteristics of a font, and are described in the *Programming Reference: Volume 2.*

Because the font metrics are so detailed, the amount of information returned to you from GpiQueryFonts may be very large. You can restrict the amount of information returned from this call in the following ways:

- You can ask for information about public or private fonts only. (Private fonts are those available only to the application issuing GpiQueryFonts.)
- You can specify an absolute number of fonts about which you require information.
- You can ask for information about fonts of a specific facename only.
- You can limit the length of the font-metrics buffer.

Some of the font metrics are illustrated in Figure 23-16 on page 23-16.

If you decide, on examining the metrics, that one of the fonts meets your requirements, the match number and the facename, when specified in the FATTRS structure of GpiCreateLogFont, are sufficient to select that font.[1] If you do not supply a facename and the presentation driver cannot find the requested font, the default font is used. If you supply a facename, the presentation driver behaves as though you had specified 0 in the match field, and proceeds as described in "Specifying 0 as a Match Value" on page 23-5.

This method of selecting a font (supplying the match number) is the one that you are recommended to use. The best way to implement a font-selection mechanism, for example, is to:

- Issue GpiQueryFonts (GSQF) to retrieve the facenames and match numbers of the available fonts.
- Organize the facenames and other information relevant to those fonts (such as point sizes) in a menu.
- When the operator has selected a font, supply the relevant information from the font metrics of that font in the FATTRS structure of GpiCreateLogFont (GSCRLF).

[1] One of the things you should consider is the resolution of the device for which the font was designed. For more information, see "Font Resolution" on page 23-6.

Specifying 0 as a Match Value

The alternative to selecting a specific font is to set the match value to 0, and complete the FATTRS structure of GpiCreateLogFont.

How a Font is Selected: If the match value is set to 0 and you have *not* set the FATTR_FONTUSE_OUTLINE indicator:

- The Presentation Manager looks for an image font that has the required selection indicators, and whose maximum baseline extent and average character width match those you have specified.
- If no suitable image font is found, the Presentation Manager looks for an outline font with the required facename and selection indicators.
- If no suitable outline font is found, the default font is made available to you.

Outline fonts obey the current character attributes (character box, shear, and angle). Because an outline font may be made available to you, even when you have requested an image font, you should issue the GpiQueryFontMetrics (GSQFM) call to find out whether the font is an image font or an outline font. GpiQueryFontMetrics returns the metrics of the current logical font. The settings of the definition indicators (field defn in the font metrics) tell you whether the font is an image or an outline font, and also whether the font is a device or a generic font. If the font is an outline font, you should specify values for the character attributes before using the font. Outline fonts are unaffected by the maximum baseline extent and average character width values you specify on GpiCreateLogFont (GSCRLF). For more information about the character attributes, see page 23-9.

Note: If you must have an image font, you should specify a match value for the font. The reason for this is that, to get an image font, you have to specify a maximum baseline extent and an average character width in the FATTRS structure. These values can be ascertained only by issuing GpiQueryFonts (GSQF), which also returns to you the match number of a font. If you specify values that are not valid, you are likely to be given an outline font.

If you have set the FATTR_FONTUSE_OUTLINE indicator:

- The Presentation Manager looks for a suitable outline font whose selection indicators match those you have specified.
- If no suitable outline font is found, the default font is used. No attempt is made to substitute an image font.

If the default font made available to you in these circumstances is an image font, and you have also specified FATTR_FONTUSE_TRANSFORMABLE, any attempt to draw graphics characters in CM_MODE3 will cause an error condition to be raised. This is an exception to the general rule that a FATTR_FONTUSE_TRANSFORMABLE font must be used in CM_MODE3 only. You can issue GpiQueryFontMetrics (GSQFM) to find out whether the current font is an image font or an outline font before you use it.

Selecting the Current Font

In addition to completing the FATTRS structure, you must supply a unique local identifier for the font on input to GpiCreateLogFont. The identifier is a number in the range 1 through 254, and is known as the font's *lcid*. As you can have more than one logical font definition in a presentation space, you have to select the current font by supplying the font's lcid on input to the GpiSetCharSet (GSCS) call.

You do not have to issue GpiSetCharSet (GSCS) to use the default font. When you want to reset the current font to its default value, however, you issue GpiSetCharSet with the value LCID_DEFAULT. The current font, like other modal attributes, reverts automatically to its default value when specific GPI calls are issued. For more information about the attribute-resetting process, see Appendix C, "Graphics Attributes."

Reassociating the Presentation Space

The match value of any device font is guaranteed unique only within the current device context. If you associate the presentation space with a different device context, any logical fonts that are also device fonts are no longer available. On the new device, the match number might identify a different device

font, which may not be suitable, or no font at all. You should, therefore, redefine a logical font, and select it as the current font, if you reassociate the presentation space when the current font is a device font. If you do not do this, the default font is used.

If you do not intend to use the font on the new device, however, you do not need to redefine it. You will be able to continue using the font if you reassociate the presentation space with the original device context.

Any generic fonts can continue to be used after the presentation space is reassociated, if they are suitable for the new device. An image font, for example, can be used only on devices that are all-points addressable.

Font Resolution

Device fonts (in particular, printer fonts) are usually designed for the resolution of the device on which they are to be used. By contrast, generic fonts are not device-specific and can be a problem because image fonts are defined for a specific device resolution. An image font designed for an EGA device, for example, might not look the same on a VGA or a printer device. Therefore, before you define a logical font, you should find out the resolution of the current device using the DevQueryCaps (WDQCAP) call with the CAPS_VERTICAL_FONT_RES and CAPS_HORIZONTAL_FONT_RES parameters. You compare the values returned from this call with the xdeviceres and ydeviceres values in the font metrics of the available fonts. These two values record the resolution of the device for which the font was designed. All measurements are in pels per inch. By comparing the font resolution with the device resolution, you can identify the image font best suited to the current device.

Deleting Logical Fonts and Unloading Physical Fonts

A logical font can be used only by the application that defines it. The logical font definition is stored in the presentation space, and is saved on the LIFO stack, together with its lcid, when a GpiSavePS (GSSPS) call is issued. Unlike a physical font definition, a logical font definition is temporary and is lost when the presentation space is deleted. To delete a logical font before then, however, you can issue the GpiDeleteSetId (GSRSS) call. This call deletes the logical font, and makes its lcid available for reuse. The logical font must not be the current font when you issue this call.

If you issue the GpiUnloadFonts (GSFUL) call to unload a private font file, you must end the association between the physical fonts in that file and the logical fonts defined by the application. You do this by:

- Issuing GpiSetCharSet (GSCS) to select a current font (such as the default font) that is not linked to the physical fonts you are unloading, and
- Issuing GpiDeleteSetId (GSRSS) to delete each logical font that references any of the physical fonts in the font resource-file. If you fail to do this, an error is returned to your application from the GpiUnloadFonts (GSFUL) call.

The Code Page

The code page is a table that assigns an ASCII or EBCDIC code point to each of the characters in a font. The assignment changes from one code page to another. Depending on the code page you select, the same code point can describe different graphics characters.

When you are using the system font, you use the GpiSetCp (GSCPG) call to select a code page. You can specify any one of the code-page identifiers returned to you from the WinQueryCpList (WMQCPL) call. The GpiQueryCp (GSQCPG) call returns the identifier of the current graphics code page for the system font. The default is code-page 850. When using any font other than the system font, you specify the required code page in the FATTRS structure of the GpiCreateLogFont (GSCRLF) call. You can find out the code page for such a font (if it is the current logical font) by using the GpiQueryFontMetrics (GSQFM) call. The available code pages are described in the *Programming Reference: Volume 2.*

Code pages are described in detail in Chapter 39, "Supporting National Languages." of this book.

Proportional Spacing and Kerning

A proportionally-spaced font is one in which some characters are allotted less space than others, so that each character occupies the proportion of space that is correct for its shape. For example, a lowercase letter *l* does not need the same space as a lowercase letter *m*. A monospaced font allows the same amount of space for each character, whatever its shape. All the Presentation Manager-supplied graphics fonts, including the system font, support proportional spacing. The font-metrics value *maxcharinc* tells you the width of the widest character in the font, and the value *avecharwidth* tells you the average width of the characters in the font.

You can also retrieve information about the widths of the characters in the current font using the GpiQueryWidthTable (GSQFWT) call. You provide the code point of the first character you are interested in, and the number of characters you want width-table information for. How the character-width information is computed is described in the *Programming Reference: Volume 2.*

Kerning, like proportional spacing, is a type of letter spacing. In a kerned font, some letters are allowed to overhang others, and therefore occupy an area that is less than the character increment for the font. This feature is restricted to those character-pairs whose appearance might benefit from greater proximity. The five character-pairs that most commonly benefit from kerning are Yo, We, To, Tr, and Ta.

Kerning is not available in the system-provided fonts. You might buy a kerned font, however, or create your own. Therefore, you can specify kerning as a requirement on the GpiCreateLogFont (GSCRLF) call. The best way to implement kerning is to use a kerning-pair table. Each of the entries in a kerning-pair table contains the code points of a pair of kerned characters, and the amount of space required between those characters. A negative value means that the space is decreased, and a positive value means that it is increased, whenever the two characters are produced together. A kerning-pair table may be provided with a purchased font. If so, its presence is recorded in the font header.

If you create an image font using the Font Editor, you can achieve a form of kerning by setting the a-space and the c-space of a character to a negative value. The a-space and the c-space are shown in Figure 23-2.

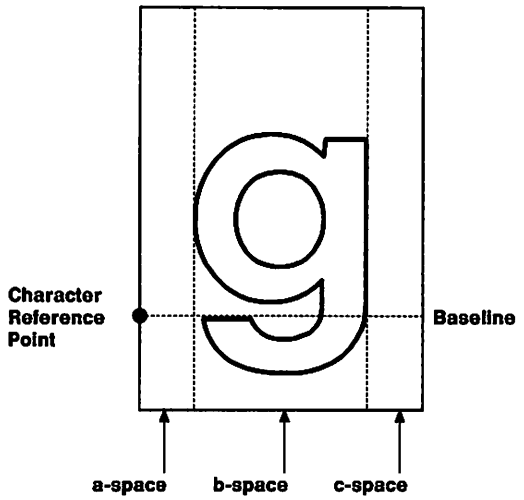


Figure 23-2. A-Space, B-Space, and C-Space

The a-space is the area of the character box before the character, the b-space is the area of the character box occupied by the character, and the c-space is the area of the character box after the character. When you specify negative a- and c-space values, they affect *any* character paired with the kerned character. Because this form of kerning cannot be applied selectively, you have to choose the kerned characters carefully. For example, an italic letter f (*f*) is a good candidate for this sort of kerning, because it can overhang most of the characters that precede and succeed it.

If kerning is specified on a character-pair basis (that is, if the font has a kerning-pair table) the font has a specific number of kerned pairs. The GpiQueryKerningPairs (GSQKPR) call returns kerning-pair information from the current logical font. For each pair of kerned characters, you are given the characters themselves, and the amount of spacing between those characters.

Kerning is not available on all devices. On devices that support kerning, kerning is enabled by default. When kerning is not supported on a particular device, kerning support is switched off by default. When kerning support is switched off, you cannot use a kerned font. To find out whether a device supports kerning, use the DevQueryCaps (WDQCAP) call.

The Character-String Primitive

To draw a string of one or more graphics characters from the current font, you use the GpiCharString (GSCHAP) call. The string starts at the current position and, when it has been drawn, the new current position is the point at which the next character would have been drawn, had there been one.[2] To draw the string starting at a position other than the current position, you use the GpiCharStringAt (GICHAR) call. This is equivalent to issuing a GpiSetCurrentPosition (GICP) or GpiMove (GIMOVE) call, followed by a GpiCharString (GSCHAP) call. You supply the text of the character string as a parameter of the appropriate GPI call.

Figure 23-3 shows some examples of the character-string primitive.

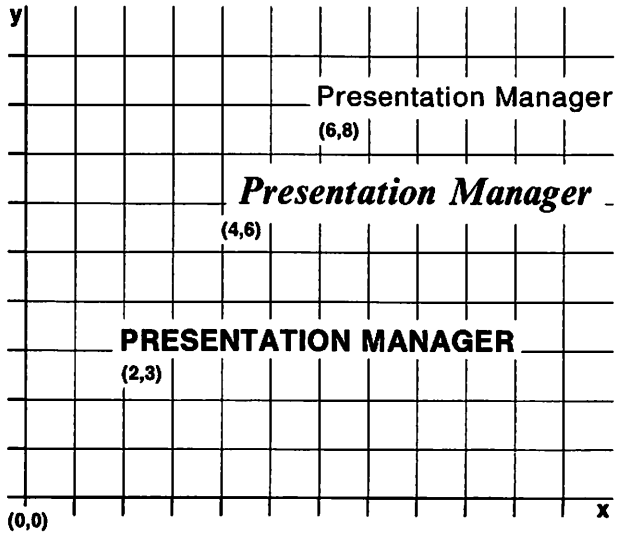


Figure 23-3. The Character String Primitive. This example shows the string *Presentation Manager* drawn from different logical fonts.

Character Modes

Every presentation space has a current *character mode*. The character mode, in conjunction with the font type (outline or image) determines the extent to which a font can be affected by the current *character attributes*. The character attributes are:

- Character box
- Character angle
- Character shear
- Character direction.

(The character attributes are described on page 23-9.)

The character mode takes effect when you draw character strings. It has no effect at the time you define a logical font. The character modes are:

CM_MODE1
An image font used when the current character mode is CM_MODE1 ignores all current character attributes, except character direction. Any outline font used when the current character mode is CM_MODE1 obeys all current character attributes. CM_MODE1 is the default character mode.

CM_MODE2
When the current character mode is CM_MODE2, the character shear, box, angle, and direction attributes are used to position characters from an image font, but the characters themselves are not sheared, rotated, scaled, or reversed. Any outline font used when the current character mode is CM_MODE2 obeys all current character attributes.

CM_MODE3
Any font used when the current character mode is CM_MODE3 *must* be an outline font. An error is raised if you try to draw a character string from an image font when the current character mode is CM_MODE3. An outline font in CM_MODE3 is affected by all current character attributes.

To select a character mode, you use the GpiSetCharMode (GSCM) call. When you have selected the mode, it applies to any font, including

[2] You can also use the WinDrawText call, which draws text to the screen a line at a time. WinDrawText is valid only in draw mode, and only to a window device context.

the system font, that is used while the mode is current.

The effects of the character modes on the character attributes are summarized in Table 23-2.

Table 23-2. How Character Attributes are Affected by Character Modes

Character Mode	Font Type	Character Attributes
MODE1	Image	Ignored
	Outline	Used
MODE2	Image	See Note 1
	Outline	Used
MODE3	Image	See Note 2
	Outline	Used

Notes:

1. Attributes are used to position the characters. The characters themselves are not sheared, rotated, scaled, or reversed.
2. An error is raised when you try to draw a character string from an image font in CM_MODE3.

Attributes of Character Strings

Each presentation space has a set of current character-string attributes. These are:

- Character box
- Character angle
- Character shear
- Character direction.[3]

The extent to which the attributes in this list affect a character string is determined by the current character mode in conjunction with the font type (image or outline). Current character attributes, like character modes, take effect when you send graphics characters to an output device. They have no effect at the time you define a logical font.

The attribute-setting calls are described individually below. In each instance, their effects in the three character modes are described.

Character Box

The character box is a rectangular boundary that defines the horizontal and vertical space occupied by a single character from a character set. All the character boxes in a single string are the same size, and the characters in the character string are positioned using the character box. The boundary of the box is not visible. The box width determines the spacing of consecutive characters along the base line, and the box height, in conjunction with the external leading value of the font, can control the line spacing of the font.

To specify the current box size, you use the GpiSetCharBox (GICB) call. As input to this call, you provide the height and the width of the character box in world coordinates.

The effects of the current character-box value vary, depending on the current character mode and font type.

The character-box value affects both the size and the position of characters drawn from an outline font, regardless of the current character mode. Each character is scaled up or down to fit the box size. This effect is shown in Figure 23-4.

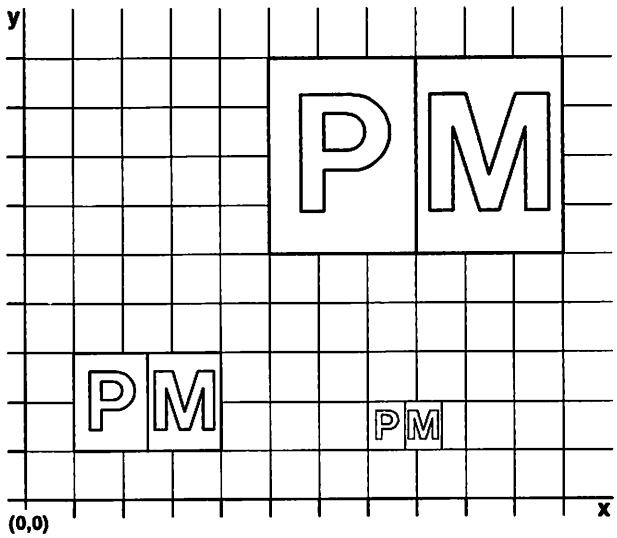


Figure 23-4. Effect of the Character Box on an Outline Font. When the character box is increased or decreased, the characters are scaled to fit the box, regardless of the current character mode.

[3] Character strings are also affected by the current color and mix attributes. These are described in Chapter 21, "The Color and Mix Attributes."

The character-box value is ignored if the current font is an image font and the current character mode is CM_MODE1, as shown in Figure 23-5.

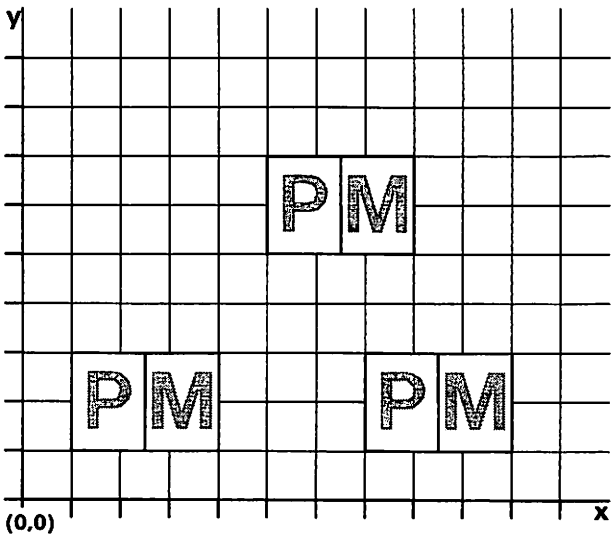


Figure 23-5. Effect of the Character Box on an Image Font in CM_MODE1. Although the character box has been both increased and decreased, the character string is unaltered.

The character-box value controls the positioning of characters from an image font when the current character mode is CM_MODE2. It cannot cause the size of each character to be scaled to fit the box. This effect is shown in Figure 23-6.

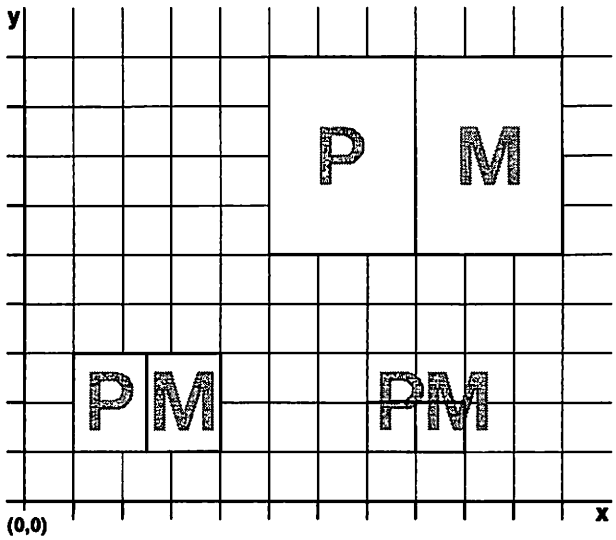


Figure 23-6. Effect of the Character Box on an Image Font in CM_MODE2. If you increase the character-box size for an image font in CM_MODE2, the characters are more widely spaced but their size is not changed. If you decrease the character-box size, the space between the characters is reduced and, because the characters themselves cannot be scaled, they may overlap.

The Default Character Box

There is a default character-box size, but it is a device-dependent value. You do not need to know what this is, except when you want to switch back to the default value, after having specified some other value. The GpiQueryDefCharBox (GIQCEL) call has been provided so that you can find out what the default is, and specify that value on a subsequent GpiSetCharBox (GICB) call. The size of the default character box is returned to you in world coordinates. The DevQueryCaps (WDQCAP) call with the values CAPS_CHAR_WIDTH and CAPS_CHAR_HEIGHT returns the size of the default character box in pels.

Character Angle

To specify the angle at which character strings are drawn, you use the GpiSetCharAngle (GICA) call. As input to GpiSetCharAngle, you supply the coordinates of the end point of a notional line to be drawn from the origin (0,0). The angle formed by the x axis of the presentation page and this notional line is the angle at which subsequent character strings are drawn. When you have specified an angle, it becomes the current setting. To reset the character angle to its default value (parallel to the x axis), you can supply zero coordinates on the GpiSetCharAngle (GICA) call.

The effects of the current character angle vary depending on the current character mode and the current font type.

When the current font is an outline font, the current character angle determines the angle of both the whole string and the individual characters in the string, regardless of the current character mode. The base line of each character box is drawn parallel to the new base line, as shown in Figure 23-7.

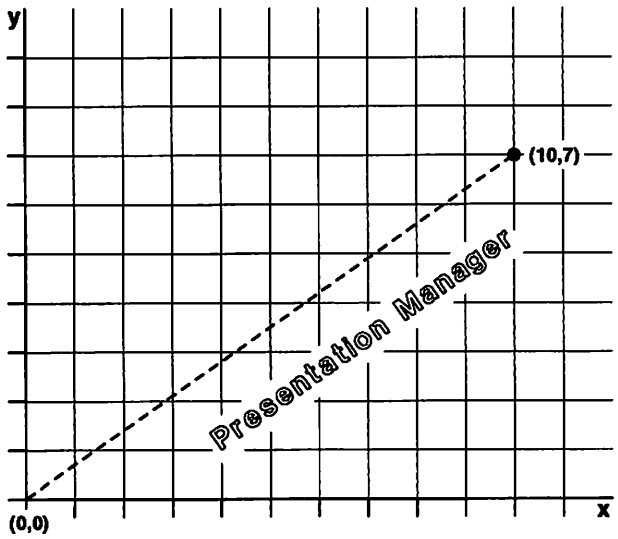


Figure 23-7. Effect of the Character Angle on an Outline Font. The character string is drawn parallel to the notional line drawn from the origin to (10,7). The base line of each of the character boxes is also drawn parallel to this notional line.

The character-angle value is ignored if the current font is an image font and the current character mode is CM_MODE1, as shown in Figure 23-8.

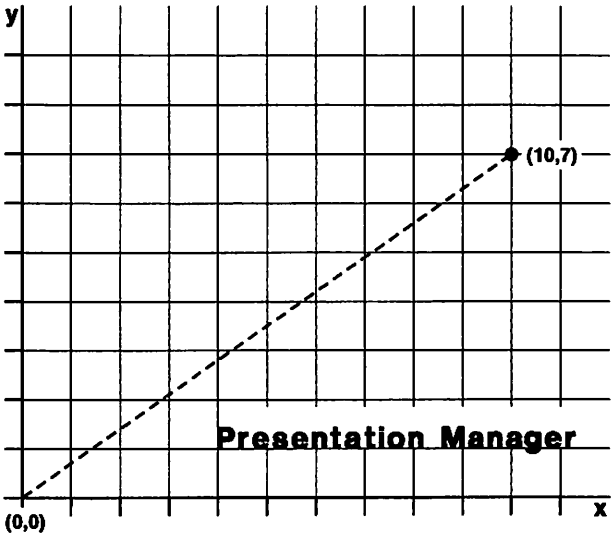


Figure 23-8. Effect of the Character Angle on an Image Font in CM_MODE1. The angle of the character string is unaltered by the GpiSetCharAngle (GICA) call.

When the current font is an image font and the current character mode is CM_MODE2, the current character angle determines the angle of the whole string, but does not affect the individual characters in the string. The *character reference point* (which is the point at which the character baseline intersects the left edge of the character box), is placed on a line that is parallel to the new base line. The base line of each character box remains parallel to the x axis. This effect is shown in Figure 23-9.

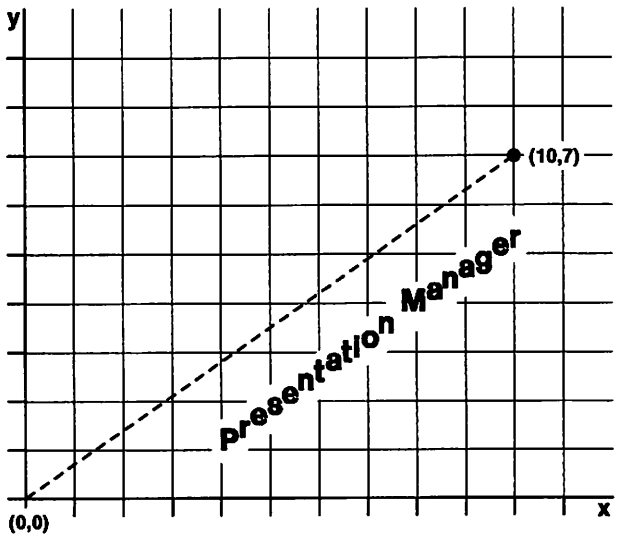


Figure 23-9. Effect of the Character Angle on an Image Font in CM_MODE2. The character string is drawn parallel to the notional line from (0,0) to (10,7).

Each of the characters in the string in Figure 23-9 is drawn with the vertical sides of its character box parallel to the y axis, and with the horizontal sides of its character box parallel to the x axis.

Character Shear

The shear of a character is the angle formed by the vertical lines of the character box, and it can affect both the positioning and the shape of the characters in the character string.

By default, the vertical lines of a character box are parallel to the y axis of the presentation page. To specify a character shear value, you use the GpiSetCharShear (GICH) call. As input to this call, you supply the coordinates of the end point of a notional line drawn from the origin (0,0). The effects of the current character shear value vary, depending on the current character mode and font type.

The character-shear value causes both the character box and the character itself to be sheared to the specified angle, if the current font is an outline font, regardless of the current character mode. This effect is shown in Figure 23-10.

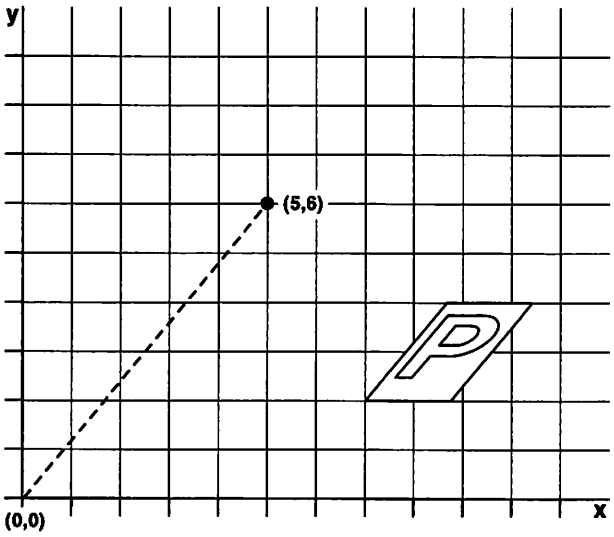


Figure 23-10. Effect of Character Shear on an Outline Font. The character box is drawn with its vertical lines parallel to the notional line from the origin to (5,6). The character is sheared to the same degree.

The character-shear value is ignored if the current font is an image font and the current character mode is CM_MODE1, as shown in Figure 23-11.

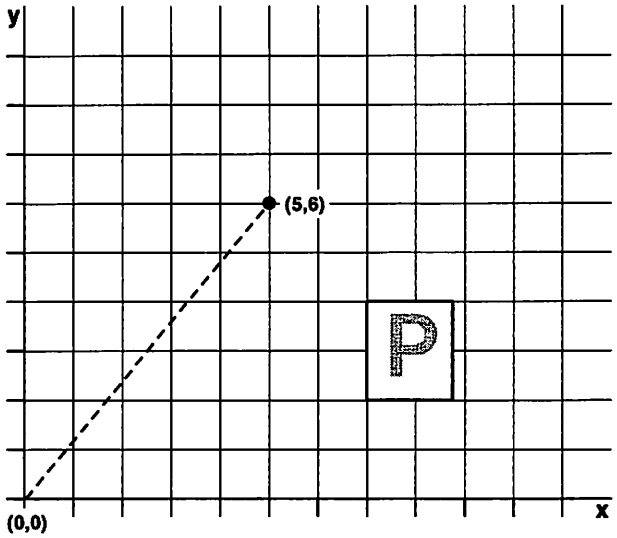


Figure 23-11. Effect of Character Shear on an Image Font in CM_MODE1. The character string is unaffected by the character-shear value.

The character-shear value affects the positioning of the characters from an image font in CM_MODE2, but the characters themselves cannot be sheared. This effect is shown in Figure 23-12.

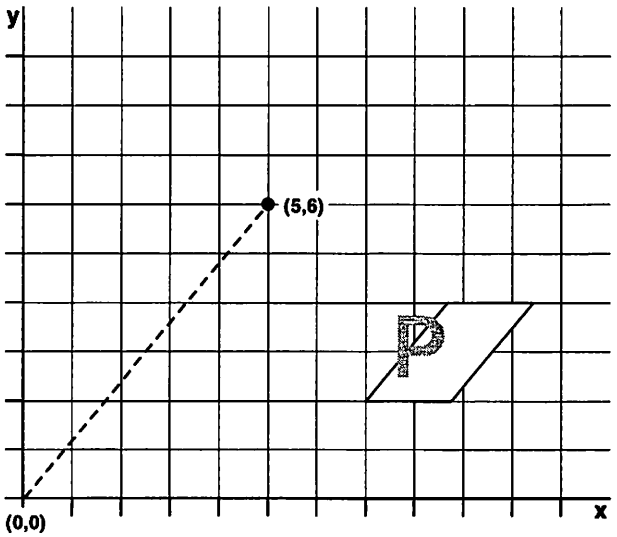


Figure 23-12. Effect of Character Shear on an Image Font in CM_MODE2. The character box is sheared, and controls the positioning of the characters, but the characters themselves are unchanged.

If the x and y values you specify on GpiSetCharShear (GICH) are both positive or both negative, the characters slope from lower left to upper right. If you supply one negative and one positive value, the characters slope from upper left to lower right.

To reset the character shear to its default value (vertical lines of the character box parallel to the y axis), you supply a coordinate value of (0,1) on GpiSetCharShear (GICH).

Character Direction

The character direction is the direction in which the characters in a string are drawn in relation to the baseline.

By default, the characters are drawn from left to right. To change the direction in which the characters of a string are drawn, you use the GpiSetCharDirection (GSCD) call. As input to this call, you supply one of the following values:

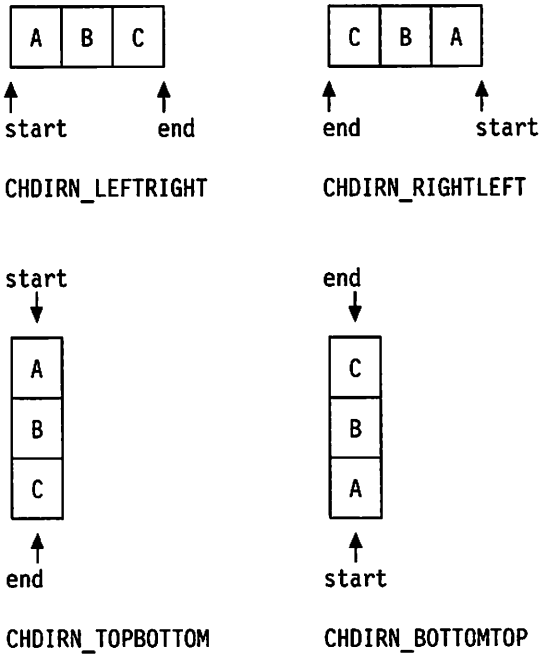


Figure 23-13. Effect of Character Direction on an Image Font or Outline Font

To query the current character direction, use the GpiQueryCharDirection (GSQCD) call.

Formatting Text

Graphics text, like all other graphics objects, has to be correctly positioned in the output area. (The output area is usually a part or the whole of the client area of a Presentation Manager window, or the addressable area of a printer.) Unlike other graphics objects, however, text is governed by well-established readability and usability rules. These rules apply generally to text output, whatever its method of production. For example, the longer the line of text, the greater the space between lines needs to be to ensure that the lines do not appear to merge. Similarly, lines of text from fonts with large point sizes need to be more widely spaced for maximum readability. Some of these considerations are taken care of by the font designer. For example, the line spacing of the font is likely to be in proportion to its point size. Other factors, such as line length, are variable. The Presentation Manager, therefore, allows you control over both the horizontal and the vertical positioning of text.

Horizontal Positioning of Text

To be able to position a character string horizontally, you need to know both the width of the output area and the length of the character string. To calculate the width of the output area, you subtract its leftmost x coordinate from its rightmost x coordinate. If the leftmost x coordinate is 30, for example, and the rightmost x coordinate is 600, the width of the output area is 570 world coordinates.

To discover the size of the character string, use the GpiQueryCharStringPosAt (GIQCRO) call, which processes the character string without actually drawing it, and without updating the current position. As input to this call, you supply a starting position for the string. You should set the x coordinate of this position equal to the leftmost x coordinate of the output area. The call returns to you an array of points, in which the world-coordinate position of each character in the string is recorded. The last position in the array is the new current position. You can also use the GpiQueryCharStringPos (GIQCPO) call, which provides the same function as GpiQueryCharStringPosAt (GIQCRO), except that you have to update the current position explicitly before issuing the call. Whichever of these calls you use, you subtract the leftmost x coordinate of the output area from the new current position to establish the length of the character string.

When a character string does not fill the width of the output area, it can be positioned in one of four ways. These are shown in Figure 23-14.



Figure 23-14. Horizontal Positioning of Text Strings

To left-align the text, simply set the x coordinate of the current position to the leftmost x coordinate of the output area before drawing the character string. To right-align the text, subtract the length of the character string from the width of the output area, and add the difference to the x coordinate of the current position before drawing the character string. If the output area is 570 world coordinates wide, for example, and the character string is 436 world coordinates long, add 134 to the x coordinate of the current position before drawing the text. To center the text, you subtract the length of the character string from the width of the output area, and then divide the difference by 2 before adding it to the x coordinate of the current position. If the difference is 134, for example, you add 67 to the x coordinate of the current position.

To justify the text, so that the text string fills the width of the output area, you distribute the surplus space throughout the character string. You could add the extra space to the break characters only, for example, or you could share it equally among all characters in the string. Text justification requires that you position each character in the string individually using either GpiCharStringPos (GICHPP) (which draws at the current position and lets you position every character after the first), or GpiCharStringPosAt (GICHPR) (which lets you position every character, including the first). Both calls allow you to specify a different character increment for each character in the string. Distance is measured from the character reference point of one character to the character reference point of the next character. The values you specify apply only to the character string you supply on input. They do *not* become current attribute values.

If the string is wider than the output area (if you are producing a whole block of text, for example), then you have to work through the string looking for break characters. The break character is usually a space, though you can check this by examining the font-metrics field usBreakChar. Each time you find a break character, you compare the length of the string (up to the break character) with the width of the output area. When the string is longer than the output area, you work back to the previous break character and display or print that part of the string. In this way, you format the whole block of text for the output area.

Vertical Positioning of Text

The line spacing of a font (also known as the *line increment*) is defined by two values recorded in the metrics of the font. These are:

- The maximum baseline extent, which defines the vertical space occupied by a character from the current font. The maximum baseline extent is the sum of the maximum ascender and the maximum descender values of the font, and is shown in Figure 23-16 on page 23-16.
- The *external leading* value, which is an additional spacing factor that is part of the font design. The Presentation Manager-supplied fonts all have an external-leading value of 0, which causes vertically contiguous character boxes to abut.

Each character in a character string is positioned by aligning its character reference point on the current position. The character reference point is the point at which the baseline of a character intersects the left edge of the character box. Thus, to position one line of text beneath another, you subtract from the character reference point of the character in line one:

- The maximum descender value of the font in line one
- The external leading value of the font in line two
- The maximum ascender value of the font in line two.

Although the two lines of text are likely to be from the same font, they do not have to be. The external leading and maximum ascender values you subtract must be those associated with the font in line two. Figure 23-15 shows the metrics involved in the horizontal positioning of text.

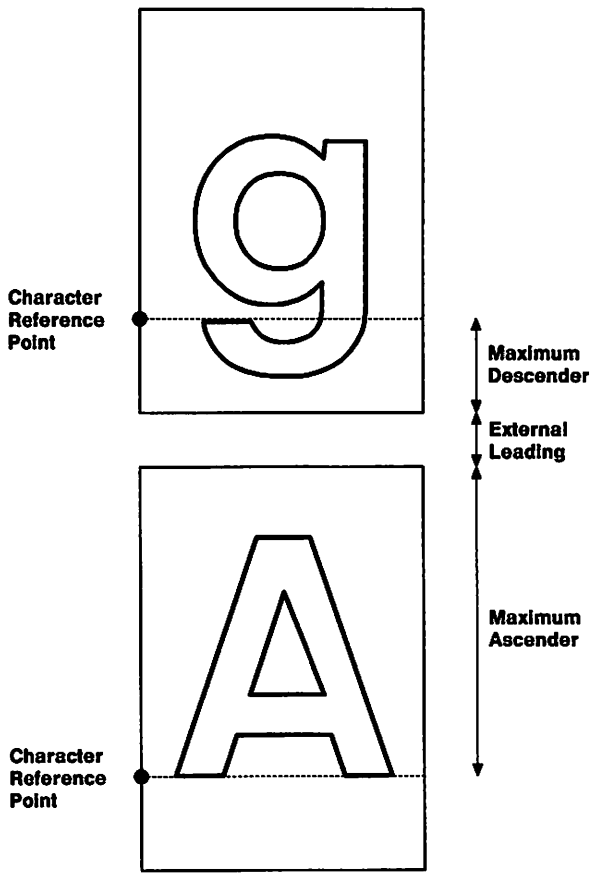


Figure 23-15. Vertical Positioning of Text Strings

If, for example, the character reference point of the first character in line one is (100,450), the maximum descender of line one is 10, the external leading of line two is 0, and the maximum ascender of line 2 is 18, then the y coordinate of the character reference point of line two is:

$450 - 10 - 0 - 18 = 422$

Therefore, the character reference point of the first character in line two should be aligned on the point (100,422).

To increase the line-spacing arrived at in this way (which you may need to do to improve readability, for example, or to fill a fixed-size output area, or to meet local line-spacing standards), you subtract the required additional increment from the y coordinate of the current position at the start of each line of text. For example, if the current position at the start of a line of text is (100,422), and you want to increase the line-spacing by two world coordinates, set the current position to (100,420) before writing the text string.

You can decrease the line spacing by borrowing space from the external leading of a font. If the external leading value is 3, for example, you can add 3 to the y coordinate of the current position at the start of a line of text. If the external leading value is already 0, you can borrow space from the *internal leading* of the font. The internal leading of a font is the space allotted for diacritics (accents) on capital letters. Internal leading is a positive value, and is recorded in the font metrics of the font. If you want to decrease the line increment by two world coordinates, for example, and you know that the internal leading can accommodate this, add 2 to the y coordinate of the current position before writing the next line of text. When you compress lines of text using this method, you must ensure that the line increment is not less than:

yMaxBaselineExt − yInternalLeading

You should also note that, with such a reduction in the line increment, accents on capital letters may interfere with the descenders of letters in the line above.

Creating Marker Sets and Pattern Sets

Like character sets, marker sets and pattern sets are defined in fonts. For both marker sets and pattern sets there are system-provided defaults (which are described on page 20-3 and on page 20-12, respectively). You can also create your own using the Font Editor. You cannot create outline markers or outline pattern sets using the Font Editor. The font definition can be loaded either as a public font (using the Install option of the Control Panel), or as a private font (using the GpiLoadFonts (GSFLO) call). The current character mode has no effect on marker sets and pattern sets. Any character from a font can be used as a marker or as an area-fill pattern.

Before you can use a locally defined marker set or pattern set, you have to issue GpiCreateLogFont (GSCRLF) to define a logical font and to give it an lcid, in just the same way as you would for a character font. Most of the values you can specify on GpiCreateLogFont are not applicable to marker sets and pattern sets. Furthermore, in the instance of marker sets and pattern sets, you generally know that you want to use a particular font.

To select a locally defined marker set or pattern set, you:

1. Issue GpiQueryFonts (GSQF) to obtain a match value for the font.
2. Issue GpiCreateLogFont (GSCRLF) specifying the facename, the match value, and an lcid for the font.
3. Select the logical font as the current marker set or pattern set by issuing GpiSetMarkerSet (GSMSS) or GpiSetPatternSet (GSPATS), as appropriate.

To revert to using the default pattern or marker sets, you issue GpiSetPatternSet (GSPATS) or GpiSetMarkerSet (GSMSS) with the value LCID_DEFAULT.

Using Bit Maps as Area-Fill Patterns

An alternative to using the Font Editor for creating your own pattern sets is to use a bit map as an area-fill pattern. (How to create bit maps is described in Chapter 24, "Bit Maps in Graphics Applications.") You have to issue the GpiSetBitmapId (GSBMID) call to give the bit map an lcid, and issue the GpiSetPatternSet (GSPATS) call specifying the lcid of the bit map. The bit map becomes a pattern set containing one pattern, and the current pattern-symbol attribute is ignored. The GpiQueryBitmapHandle (GSQBTH) call returns to you the handle of the bit map currently known by the lcid that you supply on input.

The Presentation Manager has no way of distinguishing among the lcids associated with a particular presentation space. Any lcid can identify a character set, a marker set, a pattern set, or a bit map being used as an area-fill pattern. For example, if you issue the GpiQueryNumberSetIds (GSQNSS) call to find out how many lcids are currently assigned in the presentation space, the number returned to you includes *all* logical fonts defined for this presentation space. You can also use the GpiDeleteSetId (GSRSS) call to remove any of them. If you issue this call to delete a bit map being used as an area-fill pattern, the bit map itself is not deleted. However, it is disassociated from its lcid, and the lcid can be reused.

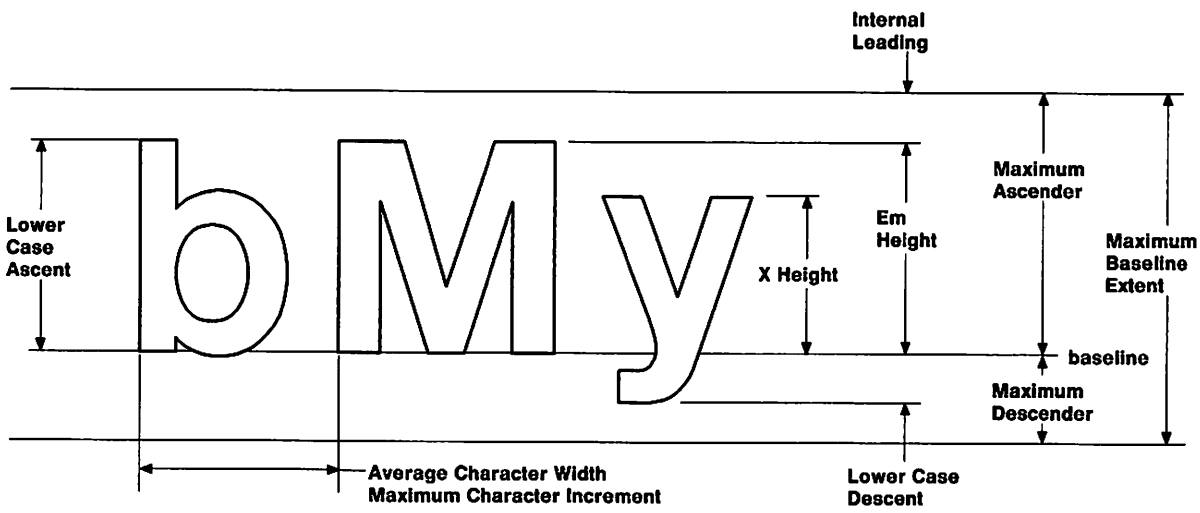


Figure 23-16. Some Font Metrics. The maximum baseline extent is the sum of the maximum ascender and the maximum descender values. The internal leading is the space allowed for diacritics above capital letters.

Example Code: Defining a Logical Font

This example includes two short extracts from the Fonttest sample program. Fonttest displays a list of the supplied fonts (Swiss, Courier, Roman, and System Proportional), and allows the user to select one. The name of the selected font is displayed in the center of the screen, and a string of text in the selected font is displayed at the bottom of the screen. Alternatively, the user can specify some of the attributes of a font interactively, and those values are used to define a logical font. In this instance, the text `User Modified Font`

is displayed in the center of the screen, and a string of text in the created font is displayed at the bottom of the screen.

Extract 1 shows how to use the GpiQueryFonts (GSQF) call to obtain the font metrics of all available fonts.

```
BOOL FonttestInitDlg(hwndDlg)
HWND hwndDlg;
{
    LONG         i;
    HPS          hps;
    LONG         NoOfFontsReq = 0;
    LONG         ActualNoOfFonts;
    FONTMETRICS  near *npMetrics;
    CHAR         Buffer[20];
    CHAR         *psz;
    CHAR         szTempText[CCHMAXSTRING];

    hps = WinGetPS(hwndClient);

    ActualNoOfFonts = GpiQueryFonts(hps,
                                    QF_PUBLIC | QF_PRIVATE,
                                    (PSZ)NULL,
                                    &NoOfFontsReq,
                                    (LONG)NULL,
                                    (PFONTMETRICS)NULL);

 /*********************************************************************/
 /* GpiQueryFonts is issued without specifying a facename and with    */
 /* the 'fonts-required' value set to 0. The purpose of this is to    */
 /* establish the total number of fonts available.                    */
 /*********************************************************************/

    sizereq = (USHORT)sizeof(FONTMETRICS) * (USHORT)ActualNoOfFonts;

    npMem = WinAllocMem(hheap,sizereq);

 /*********************************************************************/
 /* The amount of storage required for the font metrics of all fonts */
 /* is calculated and the storage is allocated from a heap.           */
 /*********************************************************************/

    npMetrics = (FONTMETRICS near *)npMem;

    GpiQueryFonts(hps,
                  QF_PUBLIC | QF_PRIVATE,
                  (PSZ)NULL,
                  &ActualNoOfFonts,
                  (LONG)sizeof(FONTMETRICS),
                  npMetrics);

 /*********************************************************************/
 /* GpiQueryFonts is issued a second time to retrieve the font        */
 /* metrics of all fonts.                                             */
 /*********************************************************************/
```

In extract 2, text in the appropriate font is displayed on the screen.

```
VOID FonttestPaint( hwnd )
HWND hwnd;
{

    HPS     hps;
    RECTL   rect;
    RECTL   PaintArea;
    POINTL  Pos;
    SIZEF   sizef;

    hps = WinBeginPaint(hwnd, (HPS)NULL, &PaintArea);

    WinFillRect(hps, &PaintArea, SYSCLR_WINDOW);

    WinQueryWindowRect( hwnd, &rect );

    WinDrawText(hps,
                strlen(szExampleText),
                (PSZ)szExampleText,
                &rect,
                SYSCLR_WINDOWTEXT,
                SYSCLR_WINDOW,
                DT_CENTER | DT_VCENTER);

  /*********************************************************************/
  /* The appropriate text string (either the name of the font or the   */
  /* string 'User Modified Font') is displayed in the center of the    */
  /* screen.                                                           */
  /*********************************************************************/


    if (Modified)
    {
        GpiSetCharMode(hps,CM_MODE2);
        sizef.cx = DisplayedFont.lAveCharWidth * 65536;
        sizef.cy = DisplayedFont.lMaxBaselineExt * 65536;
        GpiSetCharBox ( hps, &sizef );
    }

    GpiCreateLogFont(hps,
                     (PSTR8)szClassName,
                     ID_LCID,
                     &DisplayedFont);


    GpiSetCharSet(hps,ID_LCID);

  /*********************************************************************/
  /* A logical font is defined, and is selected as the current font.   */
  /*********************************************************************/

    Pos.x = 3;
    Pos.y = 5;

    WinSetPointer(HWND_DESKTOP,
                  WinQuerySysPointer(HWND_DESKTOP, SPTR_WAIT, FALSE));
```

```
    GpiCharStringAt(hps,
                    &Pos,
                    (LONG)strlen(szAlphabet),
                    (PSZ)szAlphabet);

  /*********************************************************************/
  /* A character string is drawn in the current font, beginning at the */
  /* position (3,5).                                                   */
  /*********************************************************************/

    WinSetPointer(HWND_DESKTOP,
                  WinQuerySysPointer(HWND_DESKTOP, SPTR_ARROW, FALSE));

    GpiSetCharSet(hps,0L);


    GpiDeleteSetId(hps, ID_LCID);

  /*********************************************************************/
  /* The current character set is deselected, and the current logical  */
  /* font is deleted.                                                  */
  /*********************************************************************/

    WinEndPaint( hps );
}
```

Chapter 24. Bit Maps in Graphics Applications

Screen devices are made up of a number of pels. By setting the color of the pels, you can create an image on the screen. The screen image is described internally in a *bit map*. For each pel on the screen, the bit map contains a number of bits that describe the appearance of the pel. On a monochrome device, you need only one bit to describe a pel, and that bit is simply switched on or off. Color devices need more bits to describe a single pel, because the RGB (red, green, blue) mix that gives each pel its color must be described. For example, an 8-color picture requires three bits to describe a single pel. (The use of color in graphics applications is described in Chapter 21, "The Color and Mix Attributes.")

A bit map is drawn to a screen line by line. Each horizontal line of pels is known as a *scan line*, and the bit map is retained on the screen by drawing the scan lines to the screen a given number of times a second. You are not aware of this constant refresh-activity when you look at the screen image.

The Presentation Manager supports four standard bit-map formats. These are differentiated by the number of bits allotted for the description of each pel. The standard formats are:

- 1 bit per pel
- 4 bits per pel
- 8 bits per pel
- 24 bits per pel.

For each of the standard formats, the bits that describe a screen image (the bit map) are organized in a single *bit-map plane*. The bits are stored consecutively in the bit-map plane. Thus, if you have four bits for each pel, the four bits for pel one are followed by the four bits for pel two, and so on. Pel one is at the lower-left corner of the image, and the image is scanned to the right and upward from there. The bits that describe pel one are stored beginning in the most significant bits of the first byte, and the data for each scan line is packed together. Each scan line, however, is padded at the end so that each line begins on a ULONG boundary.

The single-plane format can be converted internally to any multiple-plane format used by a device. You can also use a nonstandard number of bits to describe each pel, if your output device supports it.

If you write your own presentation driver, it should be able to convert the standard bit-map formats to its own internal format.

When to Use Bit Maps

There is no picture-generation stage to go through with bit maps, so they are most useful when speed is important. Bit maps are used as icons and pointers (where the bit map is moved frequently and quickly), as area-fill patterns, and also for graphics animation. They are especially useful for restoring the contents of a window. If you save the contents of a window in a bit map, when the window needs to be repaired (for example, when an overlying window is removed) you can restore the window contents simply by redisplaying the bit map. This is also an effective method of erasing some of the screen contents. You save the screen image in a bit map when a specific amount of drawing has been done, and later redisplay the bit map. This effectively erases any drawing done since you saved the screen contents. Bit maps are not, however, the recommended way to store graphics that are themselves going to alter, as most changes to the bit-map contents mean that you have to recreate the bit map.

A bit map created by the icon editor is saved in a device-independent form. However, the icon editor provides a facility to save individual device forms of the figure. These device forms are versions of the bit map that are tailor-created for specific display devices. For example, you may create a bit map for display on CGA, VGA, or 8514 devices. If no device-specific bit map is created, the device-independent form is used by the application. Bit maps can be produced on printers that can handle raster graphics, but they cannot be sent to devices, such as vector plotters, that are not all-points addressable. The steps involved in printing a copy of a bit map are described in "Printing a Copy of the Screen" on page 36-10.

Bit maps, in common with other graphics objects that are defined in device coordinates, cannot be scaled, translated, or rotated using the Presentation Manager transformation functions. (Transformation is described in Chapter 25, "Transformation.") There are ways of achieving some of these effects, but they may have an adverse effect on the appear-

ance of the bit map. For example, you can reduce the size of the bit map, but this requires that you copy it to a second bit map of the appropriate size and eliminate some of the original data, which may distort the image. Another potential problem you should consider is that bit maps, particularly color bit maps, can occupy large amounts of storage. The actual amount of memory occupied by a bit map is determined by both the size of the bit map and the number of bits used to describe each pel.

Creating and Drawing Bit Maps

A bit map can be created by an application, or it can be created using the Presentation Manager Icon Editor. This section begins with a description of the steps an application follows to create a bit map, and introduces the various calls available for drawing the bit map on an output device. It introduces the Icon Editor, tells you how to save and delete bit maps, and how to share them with other processes.

To create a bit map:

1. Define a *memory device context*. Like all other device contexts, this is a destination, but it is an internal one. To create the memory device context, you use the DevOpenDC (WDOPEN) call with a device type of OD_MEMORY. As input to this call you provide the device context handle of an output device with whose format this bit map must be compatible. This does not have to be the actual device to which you will be directing the bit map (although it can be), but it *must* be a device of the same type. For example, if you are directing the bit map to a printer, the device context handle must be the handle of a printer device that can print raster graphics. If you omit this value, screen compatibility is assumed.

2. Create a graphics presentation space and associate it with the memory device context. The handle of this graphics presentation space is required as input to subsequent bit-map-creation and manipulation calls.

3. Define the bit map using the GpiCreateBitmap (GSCBM) call. As input to this call, you provide the handle of the presentation space that you have associated with the memory device context. This causes the bit map to be created in a format that is compatible with the memory device context. On the GpiCreateBitmap (GSCBM) call, you must also describe the bit-map format in a *bit-map information header*. The bit-map information header has the following structure:

 Length
 : The size of the bit-map information header

 BitmapWidth
 : The number of pels in a scan line

 BitmapHeight
 : The number of scan lines

 BitmapPlanes
 : The number of bit-map planes

 BitmapCount
 : The number of adjacent bits to describe a pel in one plane.

 The output from this call is the *bit-map handle*, which you use to identify the bit map.

 To find out which bit-map formats are supported on a particular device, use the GpiQueryDeviceBitmapFormats (GSQDBF) call. This returns to you every format supported on a named device. The data is returned as an array of bit-map plane and bit-count pairs. The first pair of values in the array is the one most suitable for the device.

 You can think of the bit map at this stage as a rectangular area of memory containing random data. You can draw in the bit map at this stage by supplying (on the GpiCreateBitmap call) the address in application storage of some initialization data and setting the CBM_INIT option. This is a useful function if, for example, your application always starts by displaying the same image.

4. Select the bit map using the GpiSetBitmap (GSSBM) call. Supply both the presentation-space handle and the bit-map handle as parameters to this call. Before issuing GpiSetBitmap, you can disassociate the presentation space from the original memory device context and associate it with a different memory device context, but only if the bit-map format is convertible to a format that is supported on the new device. If you have selected one of the four standard bit-map formats, this compatibility is guaranteed and the conversion is automatic. Note that when a presentation space is associated with a memory device context, a bit map must be selected into the device context before you can draw in the presentation space.

5. Any GPI drawing requests issued to the presentation space (including those that produce graphics text) cause the bit map to receive raster images of your drawings. Before drawing in the bit map, you can clear it to CLR_BACKGROUND using the GpiErase (GSERSE) call.

When the drawing is complete, you can direct it to an appropriate output device. There are several ways to do this:

- You can use the WinDrawBitmap (WIDBMP) call. This is valid in draw mode only and only for a screen device context.
- You can use the GpiBitBlt (GIBB) call. This is more versatile than WinDrawBitmap (for example, it allows you to direct bit maps to devices other than the screen). GpiBitBlt is independent of the drawing mode, but it operates as if in draw mode. If you use GpiBitBlt, you need to define a second presentation space, and associate it with the device context of the output device. GpiBitBlt transfers the bit map from the first presentation space (the *source*) to the second presentation space (the *target*). GpiBitBlt (GIBB) is described in more detail on page 24-5.
- If you want to retain the bit-map data in the segment store of the target presentation space, you use the GpiWCBitBlt (GIWBB) call. GpiWCBitBlt is described on page 24-7.

Figure 24-1 shows the sequence of events when you create and display a bit map.

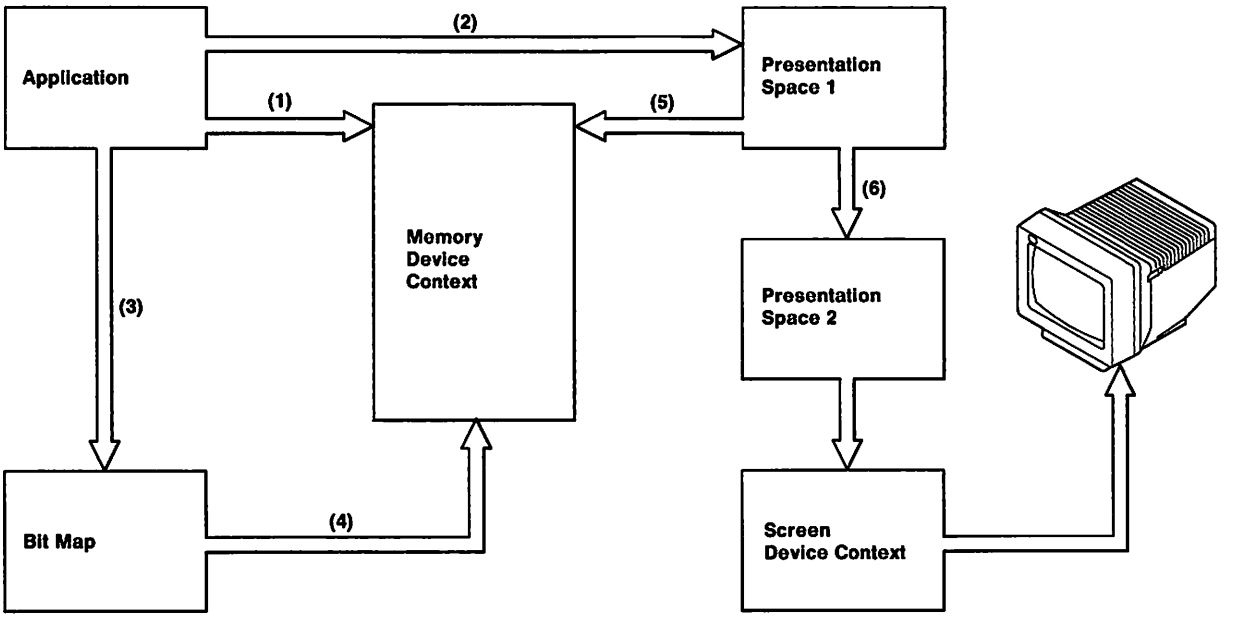


Figure 24-1. Creating and Displaying Bit Maps. The application issues the DevOpenDC call to create the memory device context (1), and then creates a graphics presentation space (2). This is associated with the memory device context. The application defines a bit map using the GpiCreateBitmap call (3). The bit map is selected into the memory device context using the GpiSetBitmap call (4). Drawing instructions issued to the presentation space cause drawing to the bit map (5). The GpiBitBlt call (6) copies the bit map from presentation space 1 to presentation space 2, which is associated with the screen device context. The bit map is transferred directly to the screen.

Creating Bit Maps Using the Icon Editor

You can use the Presentation Manager's Icon Editor to create monochrome or color bit maps that have a static appearance. The size of bit maps created in this way is in the range 1 × 1 pel to 99 × 99 pels. You can use the Icon Editor to create icons and pointers, for example, and also bit maps to use as area-fill patterns. When you use the Icon Editor to create a bit map, the bit map is saved in a resource file that can be loaded whenever it is needed.

To load a bit-map file, you use the GpiLoadBitmap call. Input to GpiLoadBitmap (GSLBM) is the identifier of the resource file containing the bit map. If you allow this value to default, the application's .EXE file is assumed to contain the bit map. Output from this call is the bit-map handle. To display the loaded bit map on the screen, you follow the sequence of steps described in "Creating and Drawing Bit Maps" on page 24-2, except that you omit steps 3 (define the bit map) and 5 (issue drawing instructions to the presentation space).

The GpiLoadBitmap (GSLBM) call lets you specify the x and y dimensions of the bit map in pels, and the loaded bit map is stretched or compressed accordingly. If you supply a zero value for one of these dimensions, the bit map is sized in the other dimension only, which is likely to cause distortion of the image. If the bit map is to be produced in its original size, specify 0 for both its width and its height.

For information on how to use the Icon Editor, see "The Icon Editor" on page 8-1.

Saving Bit Maps

To save a bit map that has been created by an application, you have two options; you can store the bit map in a *metafile* (using the GpiWCBitBlt call), or you can use the GpiQueryBitmapBits (GSQBB) and the GpiSetBitmapBits (GSBBIT) calls in conjunction with OS/2 file-handling calls. Metafiles are described in Chapter 27, "Saving and Restoring Graphics."

The GpiQueryBitmapBits (GSQBB) call copies bit-map data from a memory device context (that has a standard-format bit map selected into it) into application storage. The bit map you copy can be newly created, or it can have been loaded from a resource file.

You must supply the address in application storage of a *bit-map information table*. The bit-map information table has the same structure as the bit-map information header, except that it has an additional entry, the colors field. On input to GpiQueryBitmapBits, you supply a bit-count value and a plane value. These must both be standard bit-map format values, so the plane value is always 1, and the bit-count value is 1, 4, 8, or 24. If any conversion of the bit-map data is necessary, it is done automatically. You must also provide the address of the application storage into which the bit-map data is to be loaded. You will probably need to find out how large the bit map is before you can do this. The GpiQueryBitmapParameters (GSQBTP) call returns to you the width, height, plane count, and bit count of a named bit map.

On return from GpiQueryBitmapBits (GSQBB), the system supplies the width, the height, and the bit-map color table in the bit-map information table. You can retrieve a part of a bit map by specifying the number of the scan line from which the transfer is to start, and the number of scan lines you want.

When you have copied a bit map into application storage you can save the bit map externally, and reload it later, using OS/2 file-handling calls.

A bit map in application storage can be copied to a bit map selected into a memory device context using the GpiSetBitmapBits (GSBBIT) call. As with the GpiQueryBitmapBits (GSQBB) call, you can transfer a part of the bit map rather than the whole. GpiSetBitmapBits can be used to transfer standard-format bit maps only, and only to a bit map selected into a memory device context. If necessary, conversion of the bit-map data from one standard format to another is done automatically.

Deleting Bit Maps

It is good practice always to delete a bit map from memory when you have finished using it. To delete a bit map, use the GpiDeleteBitmap (GSDBM) call. If you have loaded a bit map from a resource file, GpiDeleteBitmap deletes only its memory version. If the bit map has not been deleted when the owning process terminates, it is deleted automatically by the system.

Making Bit Maps Available to Other Processes

When an application creates a bit map, or loads a bit map from a resource file, it can make it available to other processes by placing the bit-map handle on the clipboard. How to use the clipboard for passing data to other applications is described in Chapter 28, "User-Driven Data Exchange – The Clipboard."

The GpiBitBlt (GIBB) Call

At its simplest, GpiBitBlt (GIBB) copies a whole bit map unaltered from one device context to another. At its most powerful, GpiBitBlt (GIBB) can take a part or a whole bit map, and alter both its size and its appearance in the process of copying it to another device context. A bit map can be copied:

- From one memory device context to another memory device context.
- From a memory device context to the device context of an output device (screen window or printer) that supports raster operations.
- From the device context of a screen window to a memory device context.
- From the device context of one screen device to the device context of another screen device. In both cases, the screen device can be a Presentation Manager window.

Note: A memory device context (whether the source or the target of the GpiBitBlt operation) *must* have a bit map selected into when GpiBitBlt is issued.

GpiBitBlt (GIBB) has a number of input parameters, two of which are the handles of two presentation spaces, the source presentation space and the target presentation space. Both of these presentation spaces must be associated with an appropriate device context. Unless the associated device is a banded printer, a single presentation space can be both source and target. This allows you to replicate a bit map within a single Presentation Manager window, or to update the bit map. You could, for example, clear a part or the whole of the bit map to a single color. The effects of GpiBitBlt (GIBB) on the target bit map are controlled by the mix value, which is described in "Specifying a Mix Value."

Changing the Size of the Bit Map

You can explicitly specify the sizes of the rectangular bit-map areas in both the source and the target presentation spaces on the GpiBitBlt (GIBB) call. To do this, you provide an array of up to four device-coordinate positions on input to the GpiBitBlt call. The first two positions define the lower-left and upper-right corners of the target rectangle, and the second two positions define the same two corners of the source rectangle. If you want the two rectangles to be of equal size, do not specify the device coordinates of the upper-right corner of the source rectangle. GpiBitBlt (GIBB) automatically transfers the correct amount of data to fill the target rectangle.

If the rectangles are not to be of equal size, you have to specify the dimensions of both the source and the target rectangle, and the bit-map data is stretched or compressed to fit the target rectangle. The bit map is stretched by duplicating rows and columns of data, which may cause distortion of the image. If the data is to be compressed (that is, the source rectangle is larger than the target rectangle), you can specify one of three compression options:

BBO_OR
: means compress the bit-map data as necessary, using a logical OR operation on the eliminated rows and columns. This is most useful for white on black.

BBO_AND
: means compress the bit-map data as necessary, using a logical AND operation on the eliminated rows and columns. This is most useful for black on white.

BBO_IGNORE
: means compress the bit-map data as necessary, but ignore any eliminated rows or columns. This is most useful for color bit maps, where the results of combining pels of different colors are unpredictable.

Specifying a Mix Value

When you draw a graphics primitive into a presentation space, it is affected by the current foreground-mix value, and possibly by the current background-mix value, in that presentation space. When you use GpiBitBlt (GIBB) to copy a bit map into a target presentation space, the current foreground- and background-mix values in the target presentation space are ignored. Instead, you specify a mix-mode value on the GpiBitBlt call. The mix-mode value, also known as the *rop* (raster operation) value, determines how the bit map is affected by any data that may already be in the target presentation space, and controls the color of each pel in the bit map.

There are potentially three color settings for each pel in the bit map in the target presentation space. These are:

- The setting of that pel in the source rectangle
- The initial setting of that pel in the target rectangle
- The setting of that pel in the current area-fill pattern in the target presentation space.

The RGB values of each of these settings (taken from the *physical* color table) can be combined using Boolean operations to produce the final color of the pel in the target presentation space. As the RGB values are those from the physical color table, the results of any color mixing are unpredictable.

On input to GpiBitBlt (GIBB), you supply an index to a table of mix values. The Presentation Manager provides names for the most commonly used indexes. These are listed in the following table.

Mix Name	Effect
ROP_SRCCOPY	Source
ROP_SRCPAINT	Source OR Target
ROP_SRCAND	Source AND Target
ROP_SRCINVERT	Source XOR Target
ROP_SRCERASE	Source AND NOT (Target)
ROP_NOTSRCCOPY	NOT (Source)
ROP_NOTSRCERASE	NOT (Source) AND NOT (Target)
ROP_MERGECOPY	Source AND Pattern
ROP_MERGEPAINT	NOT (Source) OR Target
ROP_PATCOPY	Pattern
ROP_PATPAINT	NOT (Source) OR Pattern OR Target
ROP_PATINVERT	Target XOR Pattern
ROP_DSTINVERT	NOT (Target)
ROP_ZERO	0
ROP_ONE	1
ROP_GRAY	Gray pattern background

The mix value ROP_SRCCOPY is the one you are likely to use most of the time. ROP_SRCCOPY simply copies the bit map to the target presentation space without performing any color mixing. As most of the remaining mixes involve some degree of color mixing, they let you display an image that is quite different from the original.

As an alternative to the named indexes, you can supply a nonstandard index value. Possible settings of the index bits are shown in the following table.

Pattern	Source	Target (initial)	Target (final)
0	0	0	Index bit 0
0	0	1	Index bit 1
0	1	0	Index bit 2
0	1	1	Index bit 3
1	0	0	Index bit 4
1	0	1	Index bit 5
1	1	0	Index bit 6
1	1	1	Index bit 7

If you want the area-fill pattern to have no effect on the target rectangle, the 4 low-order bits of the index should be the same as the 4 high-order bits.

Converting between Monochrome and Color Data

It is possible to use GpiBitBlt (GIBB) to copy a monochrome bit map to a color bit map or device, and to copy a color bit map to a monochrome bit map or device. The Presentation Manager handles these conversions automatically according to the following rules:

- If you are copying a monochrome bit map to a color bit map or device, source 1s adopt the current image-foreground color of the target presentation space, and source 0s adopt the current image-background color of the target presentation space. For example, if image primitives in the target presentation space are blue with a yellow background, a black and white bit map becomes a blue and yellow bit map.
- If you are copying a monochrome bit map to a color bit map or device, and are allowing the current area-fill pattern color to influence the mix mode, source 1s adopt the current area-foreground color of the target device, and source 0s adopt the current area-background color of the target device.

- If you are copying a color bit map to a monochrome bit map or device, those pels that have the same color as the current image-background color in the *source* presentation space adopt the image background color of the *target* presentation space. So, for example, if the current image background color in the source presentation space is blue, all blue pels in the bit map take on the color of the current image background in the target presentation space.

 All other pels in the color bit map adopt the current image-foreground color of the *target* presentation space.

The GpiWCBitBlt (GIWBB) Call

The source and target rectangles are specified in device coordinates on the GpiBitBlt (GIWBB) call. GpiBitBlt is consequently very device-dependent, and should be avoided when creating data for interchange. Instead, you should use the GpiWCBitBlt (GIWBB) call. GpiWCBitBlt provides the same function as GpiBitBlt, with the following exceptions:

- The target rectangle is specified in world coordinates, and all four coordinates (the two source rectangle coordinates and the two target rectangle coordinates) must be specified.
- The source handle must be a bit-map handle. It must not be the handle of a source presentation space. The bit map identified by the source handle must not be selected into a memory device context when you issue GpiWCBitBlt (GIWBB).
- GpiWCBitBlt obeys the current drawing mode in the target presentation space. If the drawing mode is retain or draw-and-retain, the bit map is retained in segment store.

Single Pel Manipulation

You can manipulate single pels in a bit map using the GpiSetPel (GIPEL) and GpiQueryPel (GIQPEL) calls. The GpiSetPel (GIPEL) call sets the pel at the specified point (in world coordinates) to the current line color. This function is independent of the current drawing mode, and its effect is immediate. It is, however, a device-dependent function. The CAPS_RASTER_CAPS option of the DevQueryCaps call tells you whether the GpiSetPel (GIPEL) function is supported on the current device.

Example Code: Saving and Redisplaying Pictures in Bit Maps

This section contains two extracts from the Bmap sample program. The first is from the client window procedure (BmapWndProc). A simple picture has been drawn, and is now to be saved so that it can be redisplayed on request.

```
/*************************************************/
/* The 'Save Picture' choice has been selected.  */
/* The visible area of the picture is saved in a */
/* bit map.                                      */
/*************************************************/

hpsTemp = WinGetPS ( hwnd );

/*************************************************/
/* A cached micro presentation space is obtained.*/
/* Only the visible parts of the window rectangle*/
/* are saved. If the window rectangle is not     */
/* completely visible, a message box is displayed*/
/* so that the user can confirm the request.     */
/*************************************************/
```

```
WinQueryWindowRect( hwnd, &View );
if ( GpiRectVisible( hpsTemp, &PicRect ) != RVIS_VISIBLE )
{
   if ( WinMessageBox( HWND_DESKTOP
                     , hwnd
                     , szVisible
                     , szWindowTitle
                     , 0
                     , MB_MOVEABLE
                     | MB_CUANOTIFICATION
                     | MB_YESNO
                     ) == MBID_NO )
   {
   WinReleasePS( hpsTemp );
   return( FALSE );
   }
}
View.xLeft   = View.xLeft + 1;
View.xRight  = View.xRight - 1;
View.yBottom = View.yBottom + 1;
View.yTop    = View.yTop - 1;

View.xLeft   = (( PicRect.xLeft   < View.xLeft   )
                  ? View.xLeft   : PicRect.xLeft);
View.xRight  = (( PicRect.xRight  > View.xRight  )
                  ? View.xRight  : PicRect.xRight);
View.yBottom = (( PicRect.yBottom < View.yBottom )
                  ? View.yBottom : PicRect.yBottom);
View.yTop    = (( PicRect.yTop    > View.yTop    )
                  ? View.yTop    : PicRect.yTop);

/************************************************/
/* Define the bit-map information header and    */
/* create a bit map.                            */
/************************************************/

bmapinfo.cbFix     = 12;
bmapinfo.cx        = (USHORT)((View.xRight - View.xLeft) + 1);
bmapinfo.cy        = (USHORT)((View.yTop - View.yBottom) + 1);
bmapinfo.cPlanes   = 1L;
bmapinfo.cBitCount = 24L;

if ( hbmSimpic != NULL ) { GpiDeleteBitmap( hbmSimpic ); }
hbmSimpic = GpiCreateBitmap( hpsMem
                           , (PBITMAPINFOHEADER)&bmapinfo
                           , 0L
                           , (PBYTE)NULL
                           , (PBITMAPINFO)NULL
                           );

/************************************************/
/* The bit map is selected into the memory device */
/* context, and the source and target rectangles  */
/* of the GpiBitBlt operation are defined.        */
/************************************************/

GpiSetBitmap( hpsMem, hbmSimpic );

/***** TARGET ***********/
bmarray[0].x = 0;
```

```
bmarray[0].y = 0;
bmarray[1].x = View.xRight - View.xLeft + 1;
bmarray[1].y = View.yTop - View.yBottom + 1;

/***** SOURCE ***********/
bmarray[2].x = View.xLeft;
bmarray[2].y = View.yBottom;
bmarray[3].x = View.xRight + 1;
bmarray[3].y = View.yTop + 1;

GpiBitBlt( hpsMem
         , hpsTemp
         , 4L
         , (PPOINTL)bmarray
         , (LONG)ROP_SRCCOPY
         , (LONG)BBO_IGNORE
         );

/*************************************************/
/* When GpiBitBlt completes, the presentation    */
/* space is returned to the cache, and the       */
/* currently selected bit map is released.       */
/*************************************************/

WinReleasePS( hpsTemp );
GpiSetBitmap( hpsMem, (HBITMAP)NULL );

Current_Selection = IDMDRAWPIC;
Saved = TRUE;
PreviousTick = CurrentTick;
return ( FALSE );
```

The second extract is from the window-painting procedure (BmapPaint). It shows how a picture stored in a bit map is redisplayed.

```
case IDMBLTPIC:
    /*********************************************************/
    /* Redraw the picture that was saved.                    */
    /*********************************************************/
    WinFillRect(hps, (PRECTL)&PaintArea, CLR_CYAN);
    GpiSetBitmap( hpsMem, hbmSimpic );

    /*********************************************************/
    /* Define the source and target rectangles for the       */
    /* GpiBitBlt operation.                                  */
    /*********************************************************/
    /***** TARGET ***********/
    bmarray[0].x = rect.xLeft;
    bmarray[0].y = rect.yBottom;
    bmarray[1].x = rect.xRight;
    bmarray[1].y = rect.yTop;

    /***** SOURCE ***********/
    bmarray[2].x = 0;
    bmarray[2].y = 0;

    /*********************************************************/
```

```
/* Bitblt from the memory device context to the window.    */
/*********************************************************/
GpiBitBlt( hps
         , hpsMem
         , 3L
         , (PPOINTL)bmarray
         , (LONG)ROP_SRCCOPY
         , (LONG)BBO_IGNORE
         );

GpiSetBitmap( hpsMem, (HBITMAP)NULL );
break;
```

Chapter 25. Transformation

The transformation functions enable you to change the appearance of a graphics object *without* having to edit that object's definition. The Presentation Manager supports several types of transformation, including:

- Changes to the position of an object (the *translation* transformation)
- Changes to the size of an object (the *scaling* transformation)
- Changes to the orientation of an object (the *rotation* transformation).

You can also reflect and shear objects. In each of the transformation operations, one set of coordinate positions is replaced with another, related, set of coordinate positions.

Transformations are applied for two distinct purposes. The device transformation (which is introduced in Chapter 19, "The Graphics Presentation Space") allows you to work in any presentation-page units regardless of the target device. Its major purpose is therefore to facilitate device independence. The second type of transformation is the *modeling transformation*. There are a number of circumstances in which you need to apply modeling transformations to your graphics, and all reasonably sophisticated graphics applications are likely to make some use of them. Called segments, for example, are defined once, but they can be drawn any number of times. Because you are unlikely to want to draw a particular segment more than once in a single position, you have to be able to change the segment's position without editing the segment. The wheel of a car, for example, could be defined once but called four times. Each time, a translation transformation is applied to the original coordinates of the object, so that it appears in four different positions. Similarly, interactive graphics applications, which generally allow changes to the size and position of graphics, make full use of modeling transformations.

The Current Transformation

Every segment and primitive has a current transformation value, even if that value is simply the *identity transformation*. The identity transformation makes no change to the original coordinates of an object. The identity transformation is also referred to as the *unity transformation*. Most transformations you apply to an object can be in addition to, or instead of, the current transformation.

Coordinate Spaces

Before the mathematical principles of transformation are explained, you need to understand the way in which complex pictures are assembled. There can be up to four stages in the assembly of a picture, and at each stage there is a *coordinate space*. These are:

- World-coordinate space
- Model space
- Presentation-page space
- Device space.

The coordinate spaces are known collectively as the *viewing pipeline*. The third of these spaces contains the presentation page whose size and format you define when creating the graphics presentation space. The presentation page is a rectangle in presentation-page space. The other three spaces do not have to be defined. They are introduced to help you understand the layers of activity that may be necessary between the issue of a particular GPI drawing request, and the appearance of a drawing on the output device.

World-Coordinate Space

Each graphics object is defined separately and at a suitable precision in its own coordinate space. This is known as world-coordinate space, and the segment or primitive it contains is constructed using the coordinate positions that are coded in the object's definition. For example, if you draw a line from (8,4) to (20,75), the coordinate values (8,4) and (20,75) are world coordinates. If your presentation page is defined in GPIF_SHORT format, world-coordinate values can be anywhere in the range −32 768 through +32 767. If your presentation page is defined in GPIF_LONG format, world-coordinate values can be anywhere in the range −134 217 728 through +134 217 727. World coordinates do not have to be within the range of the presentation page. Indeed, any object that requires a high degree of detail and precision is likely to use most or all of the available coordinate range.

Figure 25-1 shows four of the world-coordinate spaces from a simplified drawing of an aircraft. These contain the outline of the body of the aircraft, a sequence of one door and several windows, the rivets from the aircraft tail, and the aircraft tail itself.

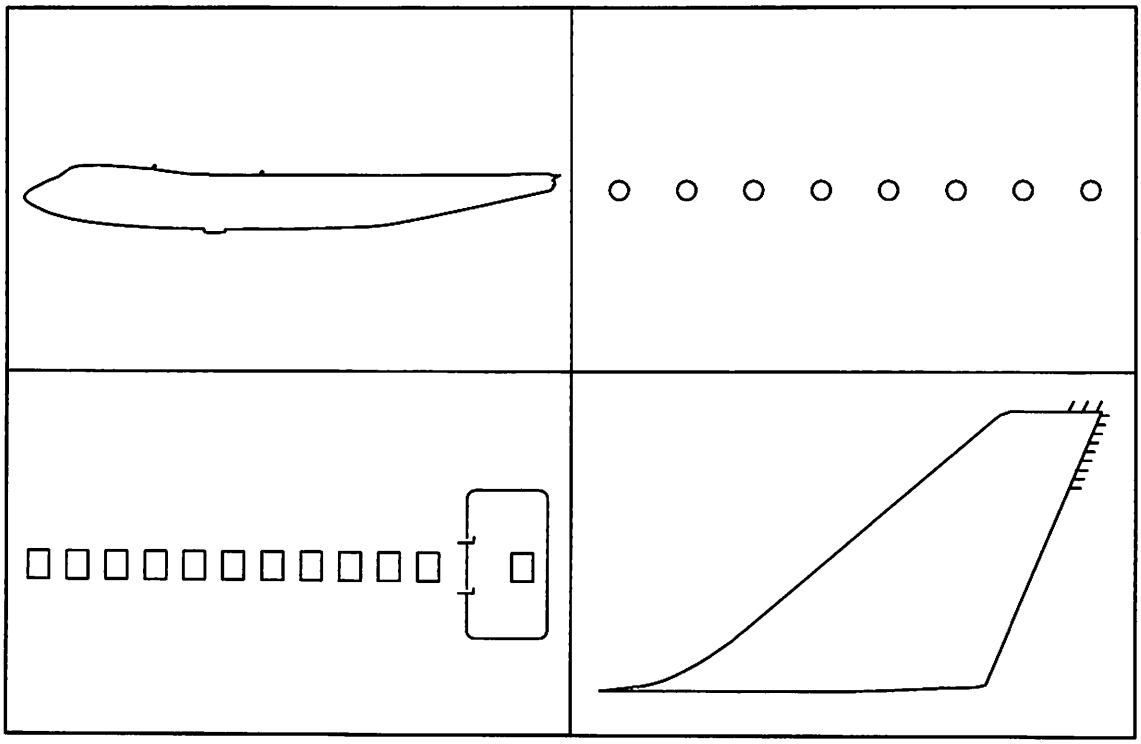


Figure 25-1. World-Coordinate Space. At this stage, the relative proportions of these objects are not apparent. Each has been defined at a suitable precision in its own world-coordinate space, and has not been restricted to the coordinate range of the presentation page. Each of these world-coordinate spaces is a graphics segment. The window and door sequence is an unchained segment that is called from another segment.

Model Space

Model space is the conceptual area where the separate components of a picture, which have been defined in world-coordinate space, are brought together. To assemble one or more world-coordinate spaces in model space, you apply a *model transformation* to each of the components. Model transformations convert world coordinates to model-space coordinates. For example, all of the parts of the aircraft drawing, which have been defined individually in world-coordinate space, are assembled in model space to produce the complete picture of the aircraft shown in Figure 25-2.

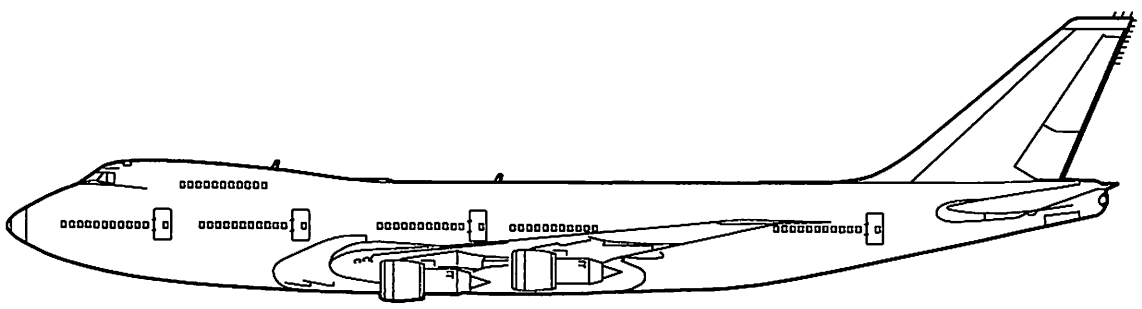


Figure 25-2. Model Space. All of the world-coordinate spaces for this drawing, including those shown in Figure 25-1, have been transformed to fit the whole.

In Figure 25-2, the door and window sequence, for example, has been called a number of times, and each time a scaling transformation and a translation transformation have been applied. The rivets have been scaled down to such an extent that they are invisible, and have been incorporated in the tail section. Model-space coordinates, like world coordinates, can be in the range −32 768 through +32 767 (in a GPIF_SHORT presentation space), or in the range −134 217 728 through +134 217 727 (in a GPIF_LONG presentation space).

The Presentation Manager identifies three distinct types of model transformation, and supplies a different call for applying each one. The three types are:

- Segment transformation
- Model transformation
- Instance transformation.

Segment Transformation

As its name suggests, the segment transformation is applied at segment level. Its major purpose is to let you draw graphics objects to a greater level of precision than is required for the complete picture. For example, in Figure 25-2 on page 25-2 a segment transformation is applied to the aircraft-tail segment to scale it down to an appropriate size before it is incorporated in the picture in model space.

To apply a segment transformation, you use the GpiSetSegmentTransformMatrix (GISTFM) call. A segment transformation can be applied only to a retained segment. You code the transformation before issuing the appropriate GpiDraw call.

You can use GpiSetSegmentTransformMatrix (GISTFM) to apply any sort of transformation to a segment. For example, you use it to translate a dynamic segment from one screen position to another when movement is requested by the operator. On receipt of the WM_MOUSEMOVE message, you:

1. Use the new mouse position to calculate the required displacement from the current position.
2. Issue GpiRemoveDynamics (GSRDYN) to remove the dynamic segment from its current position.
3. Issue GpiSetSegmentTransformMatrix (GISTFM) to translate the segment coordinates by the required amount.
4. Issue GpiDrawDynamics (GSDDYN) to redraw the segment in its new position.

Model Transformation

The GpiSetModelTransformMatrix (GISCTM) call specifies a model transformation that is to apply to subsequent primitives in the segment. You can change the model transformation any number of times in a single segment. As this call does not require you to name a segment, you can also use it to apply transformations to primitives outside segments.

Instance Transformation

Instance transformations are applied to called segments. To apply an instance transformation, you code the GpiCallSegmentMatrix (GICALM) call in the calling segment. GpiCallSegmentMatrix calls the segment and also applies the transformation that is required in this instance. For example, an instance transformation is applied to the door and window sequence each time it is incorporated in Figure 25-2 on page 25-2 to reduce its size and to position it correctly. An instance transformation applies only to the called segment, and is reset on return to the calling segment.

The following example shows a sequence of calls in which segment, model, and instance transformations are applied to graphics objects.

```
GpiSetDrawingMode (DM_RETAIN)

  /* Sets the current drawing mode        */
  /* to DM_RETAIN                         */

GpiOpenSegment (segment 1)

  /* Creates a chained segment            */

GpiCloseSegment
GpiSetSegmentAttrs                       */

  /* Make segment 1 an unchained segment  */

GpiOpenSegment (segment 2)

  /* Creates a retained, chained segment  */

GpiSetModelTransformMatrix (TRANSFORM_ADD)

  /* Specifies a transformation that is   */
  /* to apply to subsequent primitives.   */
  /* This is in addition to the current   */
```

```
  /* model transformation.                */

GpiCallSegmentMatrix (1, TRANSFORM_ADD)

  /* Calls segment 1 and applies a        */
  /* transformation to it.                */
  /* This transformation is in addition   */
  /* to the current model transformation, */
  /* and applies only to the called       */
  /* segment.                             */
   .
   .
GpiSetCurrentArcParams
GpiPointArc

  /* The 3-point arc is not subject       */
  /* to the transformation specified      */
  /* on the GpiCallSegmentMatrix call.    */
  /* The transformation that was          */
  /* current before segment 1 was called  */
  /* is applied to the remainder of       */
  /* segment 2.                           */

GpiCloseSegment
GpiSetSegmentTransformMatrix (segment 2)

  /* Specifies a segment transformation   */
  /* for segment 2                        */

GpiDrawSegment
  /* Draws segment 2                      */
```

Model space and world coordinate space are effectively the same if you are not using called segments, and if the world coordinates you use when defining each segment or primitive are exactly the same as those required in model space.

To summarize, model transformations should be used wherever possible for picture construction and modeling purposes. The segment transformation provides a way of setting the model transformation before the first element of the segment, although this can be overridden by subsequent model and instance transformations within the segment. Model transformations provide a general-purpose transformation for use within or outside segments. Instance transformations allow you to specify the size and position of a called segment. The instance transformation is reset on return to the calling segment.

Presentation-Page Space

Presentation-page space is the coordinate space in which the final picture is assembled for display or printing. There are two types of transformation operation between model space and presentation-page space. These are the *viewing transformation*, which you use to replicate model space around the display, and the *default viewing transformation*, which you apply to implement scrolling and zooming. Presentation-page coordinates can be in the range −32 768 through +32 767 (in a GPIF_SHORT presentation space) or in the range −134 217 728 through +134 217 727 (in a GPIF_LONG presentation space).

Replicating Model Space

You can draw more than one copy of a whole model space in presentation-page space, and you can transform each copy as required. You can also transform parts of the model space to the presentation page. For example, you could show an enlarged view of the tail of an aircraft, with a reduced view of the complete aircraft in one corner, as shown in Figure 25-3 on page 25-5. In this example, the picture assembled in the presentation page is derived from a single model space.

Alternatively, your displayed picture can be made up of several subpictures that have no common graphical elements. For example, you could draw the aircraft in one part of the display, and a map of an airport in another part of the display. In this instance, the final picture would be derived from different model spaces.

Whether multiple views are derived from a single model space or from different model spaces, you have two items to address for each instance of a model space that you want to incorporate into the presentation page:

1. You have to identify that part of the model space that you want to display by defining a *viewing window* for the model space.
2. You have to specify a viewing transformation to position and size the contents of the model space in presentation-page space, as required.

To get views of one or more model spaces on the screen simultaneously, you draw each model space the required number of times. Before each drawing request, you define the viewing window and specify a viewing transformation.

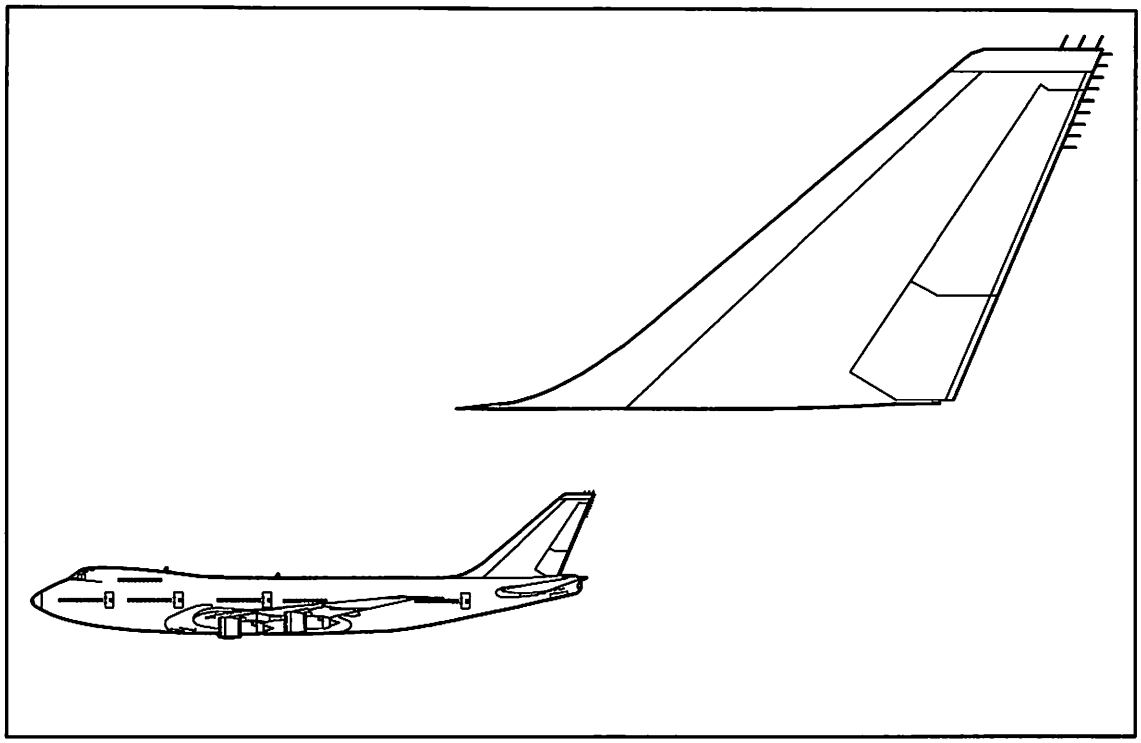


Figure 25-3. Presentation-Page Space. The whole model space (the aircraft) and a part of the model space (the tail of the aircraft) have been drawn to a single presentation-page space. In each instance, scaling and translation transformations have been applied.

Defining the Viewing Window

The viewing window is a notional boundary around the part of the model space that you are interested in. To produce the picture in Figure 25-3, the aircraft is drawn twice. The first time, the viewing window is on just the tail of the aircraft, and the second time, the viewing window is on the whole aircraft. Only those parts of the model space that are within the viewing window are visible in the picture assembled in presentation-page space.

To define the viewing window, you use the GpiSetViewingLimits (GISVL) call, which is described in Chapter 26, "Clipping." The default viewing window is the same size as the model space. Therefore, if you want to display one or more whole model spaces, you draw the picture the required number of times and allow the viewing window to default each time.

The Viewing Transformation

Each time you draw the model space, you specify a different viewing transformation to transfer a view of the picture to presentation-page space. To apply a viewing transformation, you use the GpiSetViewingTransformMatrix (GISVTM) call.

There is one viewing transformation for each part or whole model space that is to be incorporated in presentation-page space. For example, in Figure 25-3, a viewing window is specified to contain the tail of the aircraft. A viewing transformation is applied to increase the size of the tail and to position it.

GpiSetViewingTransformMatrix (GISVTM) cannot be issued while there is an open segment, and it has no effect on primitives outside segments. When it has been specified, the viewing transformation applies to all subsequently created segments until it is next changed. The viewing transformation that is current when a segment is created is a fixed part of the segment. Once specified, the viewing transformation cannot be queried, nor can it be altered, unless you recreate the segment. This is inconvenient when you are drawing several versions of the *same* picture. You can solve the problem by:

- Defining the picture in one or more unchained segments
- Creating a segment chain and calling the unchained segments from each root segment
- Setting the viewing transformation before each root segment.

Each time the segment chain is drawn, multiple views of the same picture are produced. This sequence is illustrated in the following example, where the segment chain comprises three root segments, each of which calls a single unchained segment.

```
GpiSetInitialSegmentAttrs
  /* Switches off the chained attribute      */

GpiOpenSegment
  /* Creates an unchained segment containing */
  /* the picture definition                  */

GpiCloseSegment
GpiSetInitialSegmentAttrs
  /* Switches on the chained attribute       */

GpiSetViewingTransformMatrix
  /* Sets the viewing transformation for     */
  /* segment 1                               */

GpiOpenSegment (segment 1)
GpiCallSegmentMatrix
  /* Calls the unchained segment             */

GpiCloseSegment
GpiSetViewingTransformMatrix
  /* Sets the viewing transformation for     */
  /* segment 2                               */

GpiOpenSegment (segment 2)
GpiSetViewingLimits
   /* Specify the area of interest          */
   /* in the model space.                   */

GpiCallSegmentMatrix
  /* Calls the unchained segment             */

GpiCloseSegment
GpiSetViewingTransformMatrix
  /* Sets the viewing transformation for     */
  /* segment 3                               */

GpiOpenSegment (segment 3)
GpiCallSegmentMatrix
  /* Calls the unchained segment             */
GpiCloseSegment
```

When you have created a segment chain using this method, you cannot change the viewing transformation unless you recreate the segment chain. The viewing transformation of a segment is permanently recorded and cannot be edited. You would not, however, have to recreate the picture definition in the unchained segment.

If the picture definition comprises a number of unchained segments, you should create an intermediate segment to contain the GpiCallSegmentMatrix (GICALM) calls for those segments. Each root segment would then call the intermediate segment.

The viewing transformation applies to the whole root segment, and cannot be overridden from within the segment. It is particularly useful for positioning and scaling one or more segments of a subpicture within the presentation page when a segment transformation cannot be used. (A segment transformation is overridden by any model or instance transformations within the segment.) A typical example of its use is when importing a subpicture using GpiPutData (GSPUT) or GpiPlayMetaFile (GSLOAD).

Note: The viewing transformation *must* be set to its default value before you define an unchained segment that is to be called from another segment.

Scrolling and Zooming

A default viewing transformation is applied when the screen contents are zoomed or scrolled by operator interaction. A picture is zoomed when the operator wants to increase or decrease the size of an area of interest. A picture is usually scrolled when there is more in the presentation page than can be displayed in a single page of output. Anything lying off-screen, but within the range of the presentation page, can be scrolled into view. The default viewing transformation applies to the whole of presentation-page space, and can be added to, or can replace, the current default viewing transformation. The Presentation Manager applies it after any viewing transformations.

When a presentation page is created, the default viewing transformation is set to identity. If the presentation-page contents are scrolled, for example, you:

- Erase the screen contents
- Issue GpiSetDefaultViewMatrix (GISDVM) to translate the presentation-page picture by the required amount
- Draw the picture again.

Figure 25-4 on page 25-7 shows the presentation-page contents scrolled to the left.

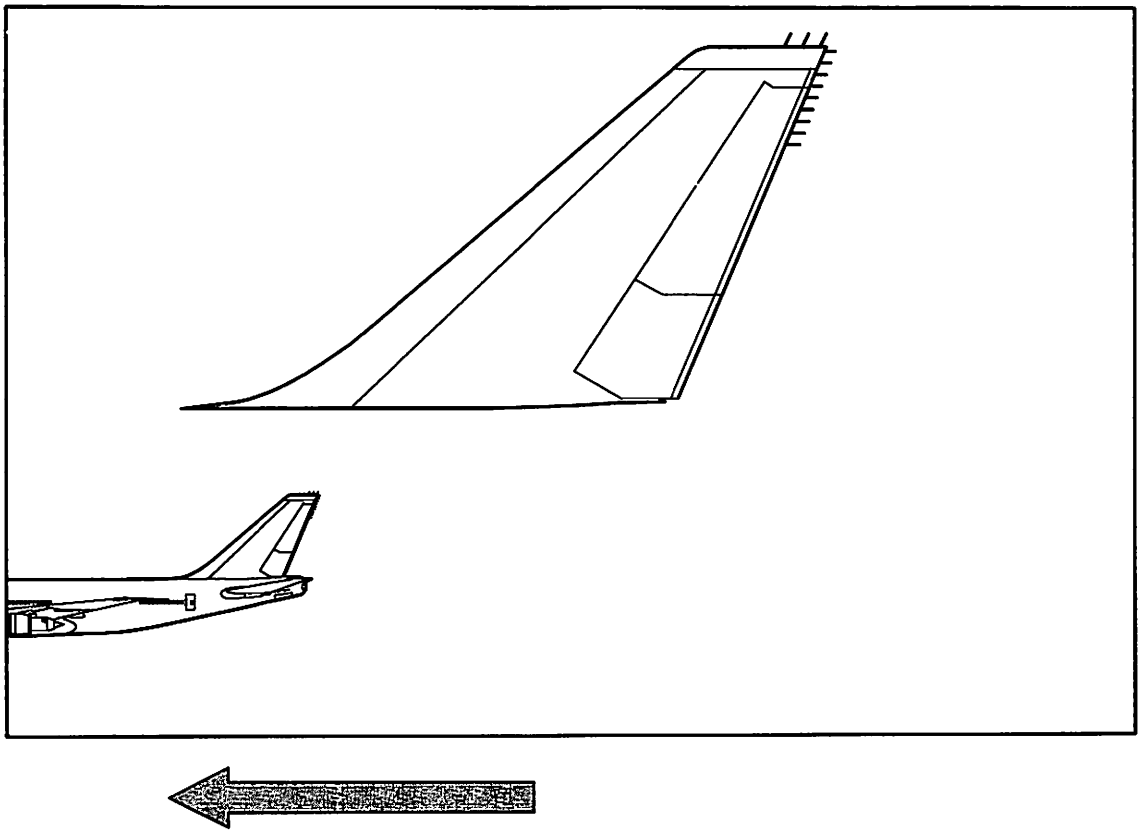


Figure 25-4. Scrolling the Presentation Page. Every presentation-page coordinate is translated to the left by the same amount.

You implement zooming in the same way, except that you use the default viewing transformation to scale the picture up or down as required.

If you want to display only one view of a single picture, and if you do not want scrolling and zooming capabilities, you can allow the viewing and default viewing transformations to default. When both transformations are allowed to default, presentation-page space is effectively the same as model space.

Whereas the model and viewing transformations can be considered a part of the picture, the default viewing transformation is part of the environment, and should not be used for picture construction. A picture is normally constructed in the presentation page with an identity default viewing transformation. You can then specify a default viewing transformation to scale and scroll the whole picture in the presentation page.

The Graphics Field

So far it has been assumed that you want to draw in the whole of the presentation-page space. You can also specify a type of viewing window for the presentation page that is smaller than the page. This window is known as the *graphics field.* To define a graphics field you use the GpiSetGraphicsField (GIFLD) call, which is described in Chapter 26, "Clipping." By default, no graphics field is specified. If you define a graphics field, the picture that you assemble within it is the picture that is visible on the output device.

Device Space

The final coordinate space in the picture-assembly process is device space. Device space is defined in device-specific units. The device transformation, which maps the picture in presentation-page space to device space, happens automatically. The device transformation is established when the presentation space is created, and ensures that graphics are displayed in the correct size and, where possible, that their aspect ratio is preserved. (The rules by which the device transformation is implemented are described in "Mapping the Presentation Page to the Device" on page 19-5.) Modify the default device transformation only if you want to use non-standard page units.

The Device Transformation

To modify the device transformation, use the GpiSetPageViewport (GIPV) call. As input to this call, you supply the device coordinates of the lower-left and upper-right corners of the page viewport.[1]

Presentation Manager automatically transforms the presentation-page contents to the area of device space within the page viewport. The drawing is not clipped to the page viewport because this is a scaling transformation only. The *whole* picture is displayed, regardless of the size of the page viewport that you specify. Figure 25-5 shows the presentation-page contents scaled to fit the page viewport.

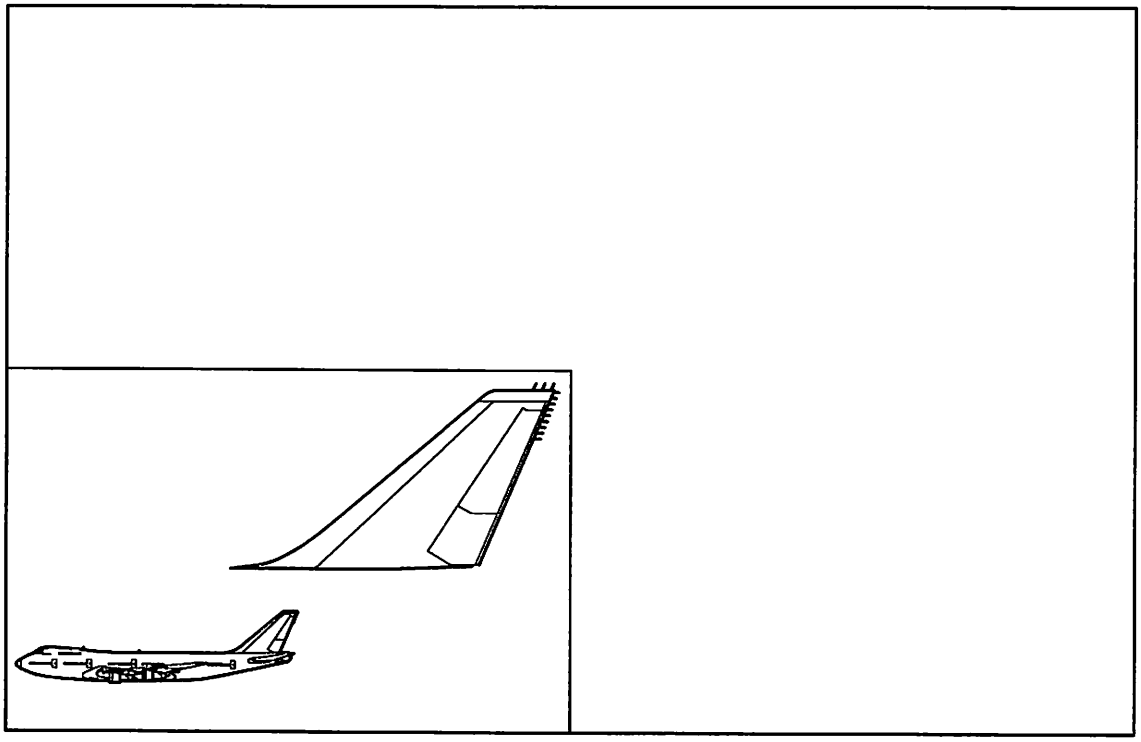


Figure 25-5. Device Space. A page viewport that is smaller than the presentation page has been defined. The picture assembled in the presentation page is therefore scaled to fit the page viewport.

[1] The page viewport is the area of the device in which the picture is displayed.

After transformation to device space, graphics coordinates must be in the range −32 768 through +32 767, even if the presentation page is defined in GPIF_LONG format. An attempt to address a coordinate outside this range may result in a coordinate-overflow error. To predict if your graphics object will give an error, do the following:

- If you are **not** rotating or shearing a graphics object, use the GpiConvert (GICONV) call to convert the device-space limits to world-coordinate-space limits. Observe these limits when you create your graphics object.
- If you are rotating or shearing a graphics object, calculate the device coordinates of the transformed object by concatenating the model-transformation, viewing-transformation, default viewing-transformation, and device-transformation matrixes (see "Concatenating Transformations" on page 25-17 and "The Device-Transformation Matrix" on page 25-19).

Remember that world-coordinate space has its own limits:

- −32 768 through +32 767 for a GPIF_SHORT-format presentation page
- −134 217 728 through +134 217 727 for a GPIF_LONG-format presentation page.

Although you can specify a page viewport of any size, the presentation page can be mapped only to an area that is equal to, or less than, the available device space. If the viewport you specify is larger than the available device space, the presentation page is mapped to an area that is the same size as the maximum available device space. To find out the dimensions of the page viewport for the currently associated device, use the GpiQueryPageViewport (GIQPV) call. You can store the dimensions of the current page viewport before changing them, and restore them later.

The Windowing-System Transformation

There is one transformation operation, the *windowing-system transformation*, that is performed automatically by the Presentation Manager. The windowing-system transformation, which is a translation transformation only, maps the coordinates of the picture in device space to the coordinates of the screen window or printer page. This happens when a picture is first drawn, and whenever the Presentation Manager window in which the picture has been drawn is moved.

Coordinate Spaces Summary

This sequence of coordinate spaces and of the transformations that link them is summarized in Figure 25-6.

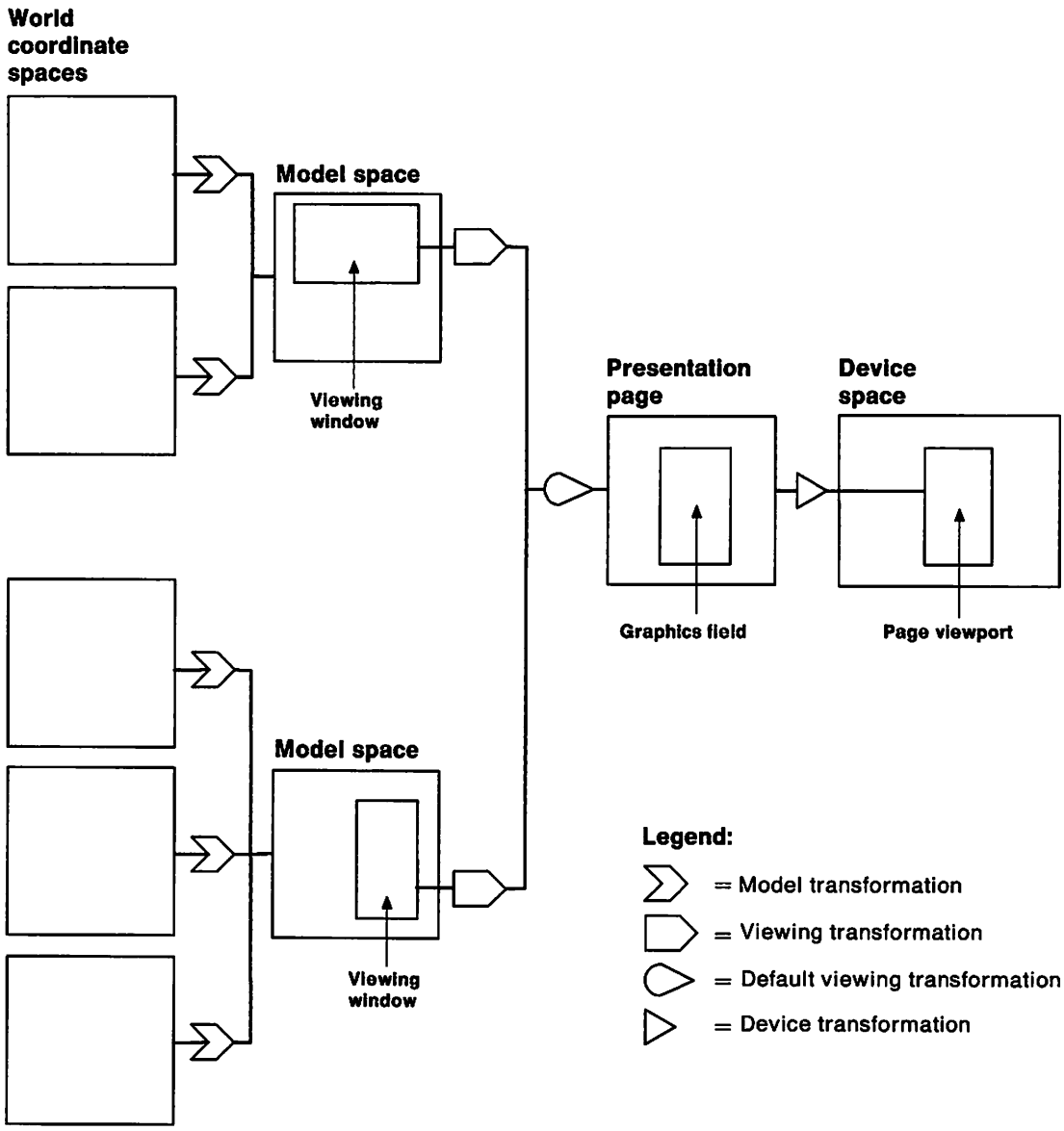


Figure 25-6. The Viewing Pipeline. By default, there is no viewing window and no graphics field. The page viewport can be allowed to default.

Transforming Bit-Map Data

As a general rule, graphics that are defined in device coordinates (bit maps and image primitives) cannot be transformed. Image primitives produced using the GpiImage (GSIMG) call, however, are an exception to this rule. The size of an image primitive is specified in device coordinates, and cannot be altered. The size therefore remains unaltered down the viewing pipeline. The *position* of an image primitive, however, is specified in world coordinates. The image is therefore subject to translation transformations.

Because the position of the image primitive is specified in world coordinates and its width is specified in device coordinates, positioning two images together on the screen causes a special problem. That is, you cannot position the second image without knowing the width, in world coordinates, of the first image. To get the width of the first image, you:

1. Identify two coordinate positions, one on the image's left edge, and one on its right edge. For example, the two positions could be (10,80) and (150,80). These positions are in device coordinates.
2. Convert these device coordinates to world coordinates using the GpiConvert (GICONV) call. GpiConvert (GICONV) converts an array of (x,y) coordinates that apply in one coordinate space to their corresponding values in another coordinate space.
3. Subtract the lower x coordinate from the higher x coordinate. In the above example, the width of the image is the difference between the world-coordinate equivalents of 150 and 10.

When you have the width of the first image in world coordinates, you can calculate the start position of the second image.

Paths, although defined in world coordinates, are device-dependent and are bound in device coordinates when they are defined. Subsequent transformations (other than the windowing system transformation) have no effect on paths. However, if a path is used to create a wide line, the width of the line is scaled as required.

Specifying Transformations

The different levels of transformation (model, viewing, and device) each have different effects on graphics, and the Presentation Manager supplies different calls to implement each one. The mathematical principles that apply to model transformations and to viewing transformations, however, are the same. The transformation equations are generally applicable in graphics applications, and you may already be familiar with them. This section provides a brief introduction to the general principles.

The transformation of an object can be represented in general terms by two linear equations that define how the (x,y) coordinates of each point in the picture are changed. The general form of these equations is:

$$x' = Ax + Cy + E$$

$$y' = Bx + Dy + F$$

where

(x, y) defines the original point
(x', y') is the transformed point
A, B, C, D, E, and F are constants.

The transformations that result from these equations depend upon the values of the constants, which in turn vary according to the type of transformation being applied. The translation, scaling, and rotation transformations are demonstrated in this chapter by applying them to the graphics object (a flag) shown in Figure 25-7.

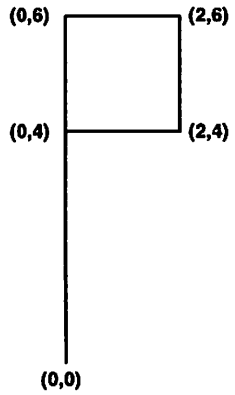


Figure 25-7. Flag. Five points define the flag. Their (x,y) coordinates are (0,0), (0,4), (0,6), (2,6), and (2,4).

Translating a Graphics Object

To move a graphics object by an absolute number of coordinate units, a translation equation is applied. The equations for translation by T_x and T_y are obtained from the general equations (with $E = T_x$, and $F = T_y$) and can be written:

$$x' = x + T_x$$

$$y' = y + T_y$$

If T_x is equal to 8, and if T_y is equal to 5, then:

$$x' = x + 8$$

$$y' = y + 5$$

The original five points are transformed:

(0,0) → (8,5)
(0,4) → (8,9)
(0,6) → (8,11)
(2,4) → (10,9)
(2,6) → (10,11)

Figure 25-8 shows the effect of translating the flag by (8,5).

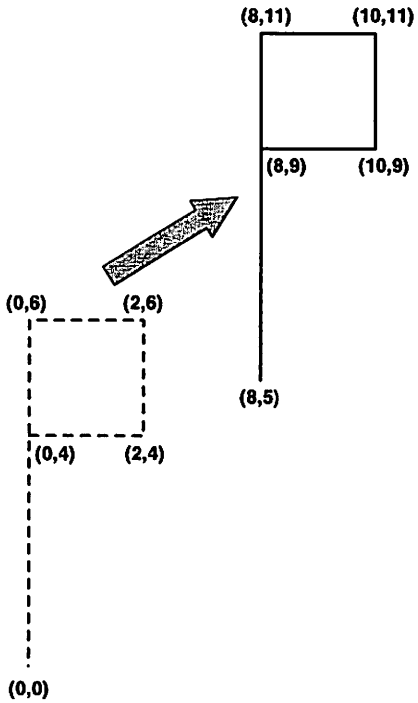


Figure 25-8. Translation by (8,5)

Translation preserves the shape, size, and orientation of the object. A negative T_x value causes movement to the left, and a negative T_y value causes movement downward.

Scaling a Graphics Object

A scaling transformation reduces or increases the size of a graphics object. A scaling factor of < 1 causes a reduction in size, and a scaling factor of > 1 causes an increase in size. Note that, if you specify a scaling factor of > 1, your graphics presentation space must be defined with the coordinate format GPIF_LONG.

The equations to scale by factors S_x and S_y are obtained from the general equations (with $A = S_x$ and $D = S_y$) and can be written:

$$x' = xS_x$$

$$y' = yS_y$$

Figure 25-9 shows the effect of scaling the flag by 0.5, by applying:

$$x' = 0.5x$$

$$y' = 0.5y$$

The original five points of the flag are transformed:

(0,0) → (0,0)
(0,4) → (0,2)
(0,6) → (0,3)
(2,4) → (1,2)
(2,6) → (1,3)

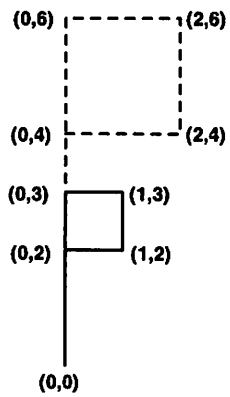


Figure 25-9. Scaling by 0.5

This scaling preserves the shape and orientation of the object, because the scaling factors in both directions are the same. However, scaling equations allow different scaling factors to be applied to x and y, which can cause distortion of the original shape of the object.

A scaling transformation reduces or increases all the coordinates of an object by the scaling factor. Any object not aligned on the x and y axes is therefore moved nearer to the origin by a reduction in size, and away from the origin by an increase in size. For example, a simple box with its four corners at (4,4), (10,4), (10,10), and (4,10) has its four corners at (2,2), (5,2), (5,5), and (2,5) if you apply a scaling factor of 0.5. To scale an object about a point

without causing it to move, a sequence of transformations is required. You:

1. Translate the scaling point of the object to the origin.
2. Scale the object at the origin.
3. Translate the scaling point of the object back to its original position.

A negative scaling factor causes a reflection of the object to be drawn. A scaling factor of -1, for example, causes a mirror image of the object to be drawn in the appropriate direction. Figure 25-10 shows the flag reflected by applying a negative y scaling factor.

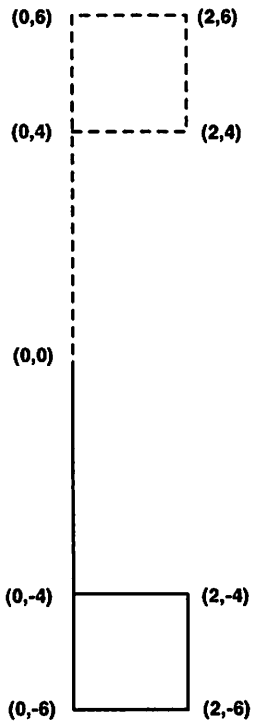


Figure 25-10. Reflection

A negative x scaling factor causes reflection in the x direction.

Rotating a Graphics Object

The equations for the rotation of an object counterclockwise about the origin (0,0), through an angle θ, are obtained from the general equations (with $A = \cos\theta, C = -\sin\theta, D = \cos\theta$, and $B = \sin\theta$) and can be written:

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

Figure 25-11 shows the effect of rotating the flag counterclockwise through 90°, by applying:

$$x' = x\cos 90 - y\sin 90$$

$$y' = x\sin 90 + y\cos 90$$

Because $\cos 90 = 0$ and $\sin 90 = 1$, these equations become:

$$x' = -y$$

$$y' = x$$

The original five points are transformed:

$(0,0) \rightarrow (0,0)$
$(0,4) \rightarrow (-4,0)$
$(0,6) \rightarrow (-6,0)$
$(2,4) \rightarrow (-4,2)$
$(2,6) \rightarrow (-6,2)$

Rotation preserves the shape and size of the object.

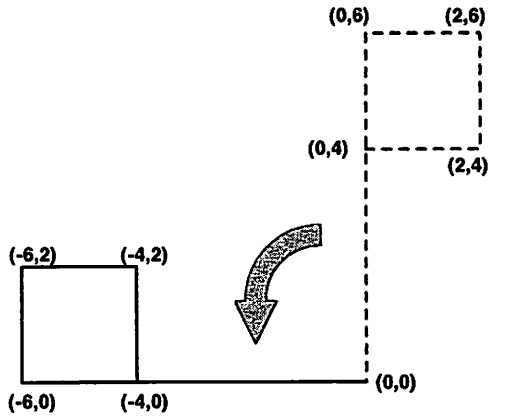


Figure 25-11. Rotation Counterclockwise Through 90°

To rotate an object around some other point (p,q), a sequence of transformations is required. You:

1. Translate the object by $(-p,-q)$ to move the point of rotation to the origin.
2. Rotate the object around the origin.
3. Translate the object by (p,q) to move it back to its original position.

A negative θ value rotates the object clockwise, and a positive θ value rotates the object counterclockwise. For clockwise rotation, the rotation equations are:

$$x' = x\cos\theta + y\sin\theta$$

$$y' = -x\sin\theta + y\cos\theta$$

Transformation Matrixes

Each of the transformation equations can be expressed more concisely in matrix format. This is simply an algebraic shorthand for the linear equations, and one that lets you request more than one type of transformation on a single GPI transformation call.

Note: Throughout this section, an * indicates matrix multiplication.

The Translation Matrix

The translation equations:

$$x' = x + T_x$$

$$y' = y + T_y$$

can be written:

$$[\,x' \;\; y'\,] = [\,x \;\; y\,] + [\,T_x \;\; T_y\,]$$

The Scaling Matrix

The scaling equations:

$$x' = xS_x$$

$$y' = yS_y$$

can be written:

$$[\,x' \;\; y'\,] = [\,x \;\; y\,] * \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

The Rotation Matrix

The equations for counterclockwise rotation:

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

can be written:

$$[\,x' \;\; y'\,] = [\,x \;\; y\,] * \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

The equations for clockwise rotation:

$$x' = x\cos\theta + y\sin\theta$$

$$y' = -x\sin\theta + y\cos\theta$$

can be written:

$$[\,x' \;\; y'\,] = [\,x \;\; y\,] * \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Combining the Transformation Matrixes

In the matrix form of the transformation equations, translation is an addition operation, whereas scaling and rotation are both multiplication operations. If they were all multiplication operations, different types of transformation could be coded on a single transformation call. For example, instead of coding three separate transformations to rotate an object around the point (p,q), you could combine all three transformations in a single call.

To overcome this problem, translation is made a multiplication operation rather than an addition operation. This requires that the vector representing a point, namely:

$$[\,x \;\; y\,]$$

be extended by a third component, w:

$$[\,x \;\; y \;\; w\,]$$

The value w is a multiplier, so that the point represented is:

$$(wx, wy)$$

In this notation, the point (x, y) is represented as:

$$[\,x\ \ y\ \ 1\,]$$

To be able to operate on such three-element vectors, and to combine translation with rotation and scaling, the 2 x 2 matrix has to be extended to 3 x 3. This is done by adding a third column of:

0
0
1

The form of the basic transformations becomes:

- For translation:

$$[\,x'\ \ y'\ \ 1\,] = [\,x\ \ y\ \ 1\,] * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}$$

- For scaling:

$$[\,x'\ \ y'\ \ 1\,] = [\,x\ \ y\ \ 1\,] * \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- For counterclockwise rotation:

$$[\,x'\ \ y'\ \ 1\,] = [\,x\ \ y\ \ 1\,] * \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The general equations:

$$x' = Ax + Cy + E$$

$$y' = Bx + Dy + F$$

become:

$$[\,x'\ \ y'\ \ 1\,] = [\,x\ \ y\ \ 1\,] * \begin{bmatrix} A & B & 0 \\ C & D & 0 \\ E & F & 1 \end{bmatrix}$$

For more information about coding more than one transformation on a single GPI call, see "Concatenating Transformations" on page 25-17.

Coding Transformations

On all GPI transformation calls, you specify the 2-dimensional transformation matrix as a 1-dimensional array in the following order:

(A,B,0,C,D,0,E,F,1)

You can specify values for more than one type of transformation on a single transformation call. The rotation values are represented by the constants *A*, *B*, *C*, and *D*; the translation values are represented by the constants *E* and *F*; and the scaling values are represented by the constants *A* and *D*.[2]

You do not have to specify each of the nine elements on every call, though those you do specify are interpreted as the first *n* of nine. If you specify the third, sixth, and ninth elements, they must be *0*, *0*, and *1* respectively.

The scaling and rotation constants (A, B, C, and D) are defined as 32-bit signed values, with a notional binary point between the second and third bytes. So, for example, 1.0 is represented as 65 536. The translation constants (E and F) are defined as 32-bit signed integers. Internally, the model, viewing, and device transformations are combined, and the resultant values are held in the same format as the individual components.

When you apply no transformations to a graphics object, it is drawn according to the identity matrix. The identity matrix looks like this:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The identity transformation causes no change to be made to the coordinates of a graphics object.

[2] You may also see the matrix elements referred to by row and column number. For example, M32 is the matrix element in the second column of row three.

Accumulating Transformations

Every graphics object, whether in world-coordinate, model, presentation-page, or device space, has a current transformation. The current model transformation is the concatenation of any instance, segment, and model transformations from the root segment downward. The current default viewing transformation is the one most recently specified. If you have not explicitly specified a transformation, the current transformation is the identity transformation. When you specify a segment transformation, a model transformation, an instance transformation, or a default viewing transformation, you specify that the transformation is to be in addition to the current transformation (TRANSFORM_ADD or TRANSFORM_PREEMPT), or is to replace the current transformation (TRANSFORM_REPLACE).

If you specify TRANSFORM_ADD, the transformation is applied after the current transformation. What this actually means is that the transformation matrix of the new transformation is post-multiplied with the transformation matrix of the current transformation. If you specify TRANSFORM_PREEMPT, the transformation matrix of the new transformation is pre-multiplied with the transformation matrix of the current transformation. This is known as a *preemptive addition.* (How transformation matrixes are multiplied together is explained in "Concatenating Transformations" on page 25-17.)

The order in which transformations are applied affects the appearance of the picture. For example, suppose you have defined a box primitive with its lower-left corner at (4,2) and its upper-right corner at (8,8), and that you want both to scale the box by 0.5 and to translate it by (−10,−10). If you translate the box before scaling it, the transformed box is as shown in Figure 25-12.

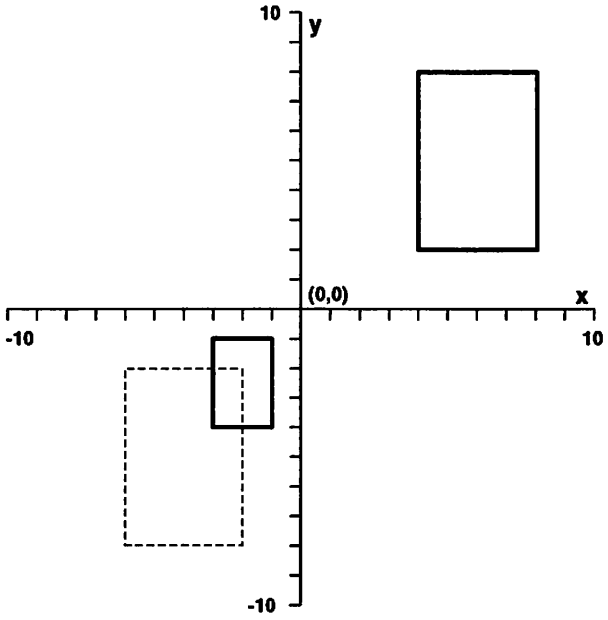


Figure 25-12. Translating before Scaling. The translated box has its lower-left corner at (−6,−8), and its upper-right corner at (−2,−2). Each of its coordinates is then scaled by 0.5, and the transformed box has its corners at (−3,−1), (−1,−1), (−3,−4), and (−1,−4).

If you scale the box before translating it, the transformed box is as shown in Figure 25-13 on page 25-17.

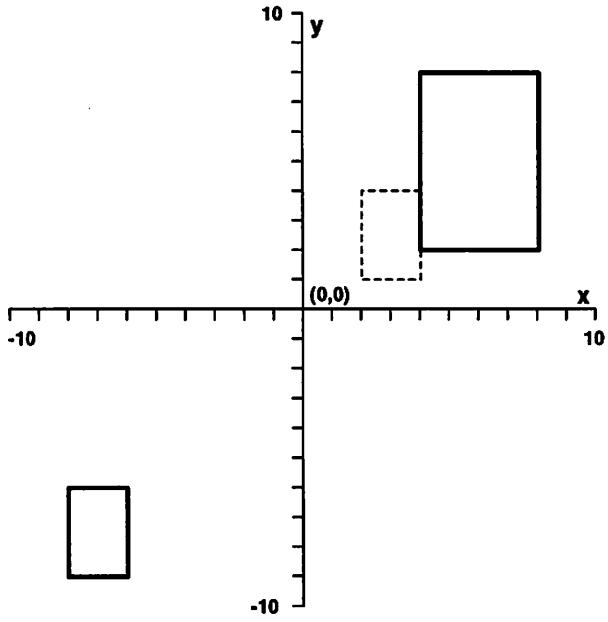


Figure 25-13. Scaling before Translating. The scaled box has its lower-left corner at (2,1), and its upper-right corner at (4,4). The box is then translated by (−10,−10), and the transformed box has its corners at (−8,−6), (−6,−6), (−6,−9), and (−8,−9).

When you are drawing a picture in which there are called segments, and in which transformations are applied to the root segments, you usually want to ensure that the root-segment transformations are applied to any segments they call. For example, if a segment that is translated to the left of the picture (by changing its segment transformation) calls a second segment, that leftward transformation should also be applied to the called segment. In this instance you would specify TRANSFORM_ADD on the GpiCallSegmentMatrix (GICALM) call to add the instance transformation to the calling segments' segment transformation. Instance transformations are automatically reset on return to the calling segment.

Concatenating Transformations

When you apply more than one transformation to a single object, you can apply the transformations consecutively, or you can *concatenate* them on a single call. Concatenation is better for performance than a series of individual transformations. It is most useful when the transformations are applied to obtain a single effect, such as a rotation around the point (p,q), or a change to the size of an object without changing the object's position.

To concatenate transformations, you multiply together the individual transformation matrixes. The product of this multiplication is the concatenated transformation. Three helper functions, GpiTranslate (GSXLAT), GpiRotate (GSROT), and GpiScale (GSSCAL), are provided to perform the matrix arithmetic required to concatenate transformations. These helper functions merely calculate the appropriate matrix; the transformation is applied to graphics segments by passing this matrix to the Gpi call which sets the appropriate transform.

To show how matrix multiplication is implemented, here are two matrixes:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad * \quad \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix}$$

The multiplication of these two matrixes produces a value for each of the nine elements of the resulting matrix:

$$\begin{bmatrix} \textit{element}\ 1 & \textit{element}\ 2 & \textit{element}\ 3 \\ \textit{element}\ 4 & \textit{element}\ 5 & \textit{element}\ 6 \\ \textit{element}\ 7 & \textit{element}\ 8 & \textit{element}\ 9 \end{bmatrix}$$

To produce element 1 of the matrix, element a is multiplied by element j, element b is multiplied by element m, and element c is multiplied by element p. That is:

$$\textit{element}\ 1 = (a \times j) + (b \times m) + (c \times p)$$

To produce element 2 of the matrix, element a is multiplied by element k, element b is multiplied by element n, and element c is multiplied by element q. To produce element 3 of the matrix, elements a, b, and c are multiplied by their corresponding elements (l, o, and r) in the third vertical line of the second matrix. To produce element 4 of the matrix, you move down a row in the first matrix. That is, element d is multiplied by element j, element e is multiplied by element m, and element f is multiplied by element p. You continue the multiplication in this way until each of the nine elements has a value. The complete workings for this example are as follows:

$$element\ 1 = (a \times j) + (b \times m) + (c \times p)$$

$$element\ 2 = (a \times k) + (b \times n) + (c \times q)$$

$$element\ 3 = (a \times l) + (b \times o) + (c \times r)$$

$$element\ 4 = (d \times j) + (e \times m) + (f \times p)$$

$$element\ 5 = (d \times k) + (e \times n) + (f \times q)$$

$$element\ 6 = (d \times l) + (e \times o) + (f \times r)$$

$$element\ 7 = (g \times j) + (h \times m) + (i \times p)$$

$$element\ 8 = (g \times k) + (h \times n) + (i \times q)$$

$$element\ 9 = (g \times l) + (h \times o) + (i \times r)$$

Note that if the order of the two matrixes is reversed, the results of the multiplication are different.

Here is a simple example in which an object is scaled by a factor of 3, and then translated by (5,4):

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 5 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 5 & 4 & 1 \end{bmatrix}$$

You can multiply together as many transformation matrixes as you require.

A simple example of using transformation matrixes to change the position of an object is the GpiTranslate (GSXLAT) call. As input to GpiTranslate, you specify the co-ordinates of the point to which you want to move the object and the name of the transform matrix that you want to use. The transform matrix should be in the form of a one-dimensional array. You can also specify whether this transformation should replace the value for a previous transformation, or whether it should be added to it.

A more complex example of using matrixes is to rotate an object counterclockwise about the point (p,q) using one transformation call. This requires three transformations to be concatenated. You can do this in one of two ways:

- You can use the GpiRotate (GSROT) call.

 When you use the GpiRotate call, you specify the angle of rotation, the coordinates of the point around which you want to rotate the object, and the transform matrix. The transform matrix should be in the form of a one dimensional array. You can also specify whether this transformation should replace the value for a previous transformation, or whether it should be added to it.

 If you specify that the current transformation should be combined with a previous transformation, the matrix for the concatenated transformation is produced incrementally, as described below.

- You can specify each transformation step by step. The sequence of actions would be:

 1. Translate the object by (−p,−q) to move the point of rotation to the origin
 2. Rotate the object about the origin
 3. Translate the object by (p,q) to move it back to its original position.

 This method is more suitable for complex transformations.

The individual matrixes are:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -T_x & -T_y & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}$$

The matrix for the concatenated transformation is produced incrementally. That is, two adjacent matrixes are multiplied to produce a single matrix, which is then multiplied with the third matrix. You can begin by multiplying either the first two matrixes or the second two matrixes. If you start by multiplying matrixes two and three together, the resulting matrix is:

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ T_x & T_y & 1 \end{bmatrix}$$

This matrix is multiplied with the first matrix to produce the matrix for rotating an object at the point (p,q):

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ a & b & 1 \end{bmatrix}$$

where:

$a = (-T_x \cos\theta + T_y \sin\theta + T_x)$

and

$b = (-T_x \sin\theta - T_y \cos\theta + T_y)$

Similarly, to scale an object at a point without also moving the object, you can:

- Use the GpiScale (GSSCAL) call.

 When you use the GpiScale call, you specify the scaling factor, the coordinates of the center point, and the transform matrix. The transform matrix should be in the form of a one-dimensional array. You can also specify whether this transformation should replace the value for a previous transformation, or whether it should be added to it.

- Specify the transformation step by step. The sequence of actions would be:

 1. Translate the object's scaling point to the origin
 2. Scale the object at the origin
 3. Translate the object back to its original position.

Here are the three matrixes required to obtain this effect:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -T_x & -T_y & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}$$

The matrix of the concatenated transformation is:

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ (-T_xS_x + T_x) & (-T_yS_y + T_y) & 1 \end{bmatrix}$$

The Device-Transformation Matrix

To use the device-transformation matrix you need to know the following values:

X_1 the x coordinate of the lower-left corner of the page viewport

Y_1 the y coordinate of the lower-left corner of the page viewport

X_2 the x coordinate of the upper-right corner of the page viewport

Y_2 the y coordinate of the upper-right corner of the page viewport

X_{PS} the presentation-space width −1

Y_{PS} the presentation-space height −1.

The device-transformation matrix is defined as:

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ c & d & 1 \end{bmatrix}$$

where:

$a = (X_2 - X_1)/(X_{PS}+1)$

$b = (Y_2 - Y_1)/(Y_{PS}+1)$

$c = X_1 + (a-1)/2$

$d = Y_1 + (b-1)/2$

Shearing a Graphics Object

A shearing transformation alters the shape of an object by translating its x coordinates relative to its y coordinates, or its y coordinates relative to its x coordinates. The amount by which the coordinates are translated is determined by the angle of the shear. Figure 25-14 shows the flag sheared to the left along the x axis.

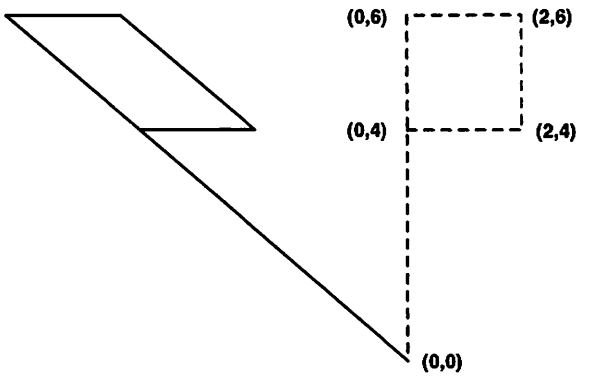


Figure 25-14. Shearing Along the x Axis. In this example, the y coordinates are unaltered.

The equation for shearing an object to the left along the x axis by angle θ is:

$$x' = x - y \sin \theta$$

$$y' = y$$

The matrix form of the equation is:

$$\begin{bmatrix} 1 & 0 & 0 \\ -\sin\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To shear an object along the y axis, the angle of the shear is represented by constant B in the general equation.

Example Code: Changing the Segment Transformation

This example, which is taken from the Graphic1 sample program, shows two methods of updating a segment transformation. The first is used when the segment is unchained and is called from another segment, in which case its segment transformation is coded on the GpiCallSegmentMatrix (GICALM) call. The second is used when the segment is chained.

```
VOID Translate()
{
  POINTL ptlOldWork, ptlNewWork;

  if (ptlOldMouse.x == ptlNewMouse.x &&
      ptlOldMouse.y == ptlNewMouse.y){
      return;
  } /* endif */
  ptlOldWork = ptlOldMouse;
  GpiConvert(hpsClient, CVTC_PAGE, CVTC_MODEL, (LONG)1, &ptlOldWork);
  ptlNewWork = ptlNewMouse;
  GpiConvert(hpsClient, CVTC_PAGE, CVTC_MODEL, (LONG)1, &ptlNewWork);

  /*********************************************************************/
  /* If the new position of the segment is different from the current  */
  /* position of the segment, the current position is stored. Both the */
  /* current position and the new position are converted to model-     */
  /* space coordinates.                                                */
  /*********************************************************************/
```

```
if (!fChained){
    GpiSetEditMode(hpsClient, SEGEM_REPLACE);
    GpiOpenSegment(hpsClient, ulRootSeg);
    GpiSetElementPointerAtLabel(hpsClient, SEGCALL);
    GpiOffsetElementPointer(hpsClient, (ULONG)1);
    amatlfSegXform[ulRootSeg].lM31 += (ptlNewWork.x-ptlOldWork.x);
    amatlfSegXform[ulRootSeg].lM32 += (ptlNewWork.y-ptlOldWork.y);
    GpiCallSegmentMatrix(hpsClient,
                         ulEditSeg,
                        (ULONG)9,
                        &amatlfSegXform[ulRootSeg],
                        TRANSFORM_ADD);
    GpiCloseSegment(hpsClient);

/*********************************************************************/
/* If the segment is a called segment, the GpiCallSegmentMatrix call */
/* in the calling segment has to be updated. The edit mode is set to */
/* 'replace', the segment is opened, and the label SEGCALL is        */
/* located. The element pointer is incremented by 1 to locate the    */
/* call that is being replaced. The current transformation value is  */
/* updated, and is specified on the new GpiCallSegmentMatrix call.   */
/* The edit is ended by closing the segment.                         */
/*********************************************************************/

} else {
    GpiQuerySegmentTransformMatrix(hpsClient,
                                   ulRootSeg,
                                  (ULONG)9,
                                  &amatlfSegXform[ulRootSeg]);
    amatlfSegXform[ulRootSeg].lM31 += (ptlNewWork.x-ptlOldWork.x);
    amatlfSegXform[ulRootSeg].lM32 += (ptlNewWork.y-ptlOldWork.y);
    GpiSetSegmentTransformMatrix(hpsClient,
                                 ulRootSeg,
                                (ULONG)9,
                                &amatlfSegXform[ulRootSeg],
                                 TRANSFORM_REPLACE);

/*********************************************************************/
/* If the segment is chained, the current segment transformation is  */
/* queried and updated. It is then respecified using                 */
/* GpiSetSegmentTransformMatrix.                                     */
/*********************************************************************/

} /* endif */
```

Chapter 26. Clipping

Clipping is a process that discards the parts of a picture that lie outside a specified *clipping boundary*, and it can occur at each stage in the viewing pipeline. That is, clipping can occur in world-coordinate space, in model space, in presentation-page space, and in device space. The clipping boundaries are:

The Clipping Path
The clipping path is a boundary defined in world-coordinate space to which subsequent primitives are clipped. The clipping path can be a nonrectangular boundary.

The Viewing Window
The viewing window is defined in model space. It outlines those parts of model space that you want to be visible when model space is transformed to presentation-page space. The viewing window is a rectangular clipping boundary. Any drawing outside it is clipped away.

The Graphics Field
The graphics field is defined in presentation-page space. It outlines the part of the presentation page that is to be visible when presentation-page space is transformed to device space. The graphics field is a rectangular clipping boundary. Any drawing outside it is clipped away.

The Clipping Region
The clipping region is defined in device space. The boundary of the clipping region can be a single rectangle, or it can comprise a number of rectangles that either overlap each other or that are disjoint. The boundary's purpose is to let you redraw a part of a display after operator interaction, for example, by defining the part of the display that needs to be redrawn, and clipping away the parts of the picture that are unaltered.

The clipping boundaries are application boundaries, and are quite separate from those defined by the windowing process, where output is clipped to the visible region of the window. When you specify no clipping boundaries, graphics are clipped to the intersection of the presentation page and the client area of the Presentation Manager window. The presentation page itself is not a clipping boundary.

The next section of this chapter explains the rules that determine how a given object is clipped. Each of the clipping boundaries is then described in detail.

How Clipping is Implemented

The rules by which the Presentation Manager implements clipping are:

- Any primitive that is completely outside the clipping boundary is discarded.
- Any primitive that is completely within the clipping boundary is retained.
- When a primitive crosses the clipping boundary, any part of it that is outside the boundary is discarded. Any part that is within the boundary is retained. When the clipping boundary is a clipping path, a viewing window, or a graphics field, any point that falls on the boundary is considered to be within the clipping boundary. When the clipping boundary is a clipping region, any point that is on the top or right boundary of a rectangle is discarded, and any point that is on the bottom or left boundary of a rectangle, that is not also on the right or top boundary, is included in the region.

The Clipping Path

How to define a path, and how to use a path to construct wide lines and filled shapes, are described in Chapter 20, "Graphic Primitives and Their Attributes." This section gives some additional general information about paths, and describes how to use them as clipping boundaries.

A path definition can be stored in a graphics segment. If the current drawing mode is retain, however, the path is not defined, and does not therefore become the current path, until that segment is drawn.

There are advantages in retaining a path definition in a graphics segment. For example, the path definition can be edited and redrawn. The ability to recreate a path definition becomes especially useful when you reassociate the presentation space. Although the primitives that constitute the path definition are defined in world coordinates, the path

is constructed in device coordinates and is held in the current device context. If you associate the presentation space with a different device context, you have to recreate the current path definition to continue using it. If the path definition has been retained in segment store, you can recreate it simply by redrawing the appropriate segment. Similarly, a path definition is lost after a single GpiFillPath (GSFILP) or GpiStrokePath (GSSTRP) call, and also when superseded as the current clipping path. If its definition is retained in segment store, the path can be recreated by redrawing the segment.

Using the Path as a Clipping Boundary

The path that you use for clipping purposes can be modified using GpiModifyPath (GSMODP) (so that the path is converted using the current line-width, line-join, and line-end values), or it can be a simple unmodified path. Before the current path can be used as a clipping boundary, it must be selected as the current clipping path. To do this, you use the GpiSetClipPath (GSSCLP) call. If there is already a current clipping path defined for this presentation space, the new clipping path is intersected with it. When you have issued GpiSetClipPath, all subsequent drawing is clipped to the path.

Figure 26-1 shows a simple example of a path used as a clipping boundary.

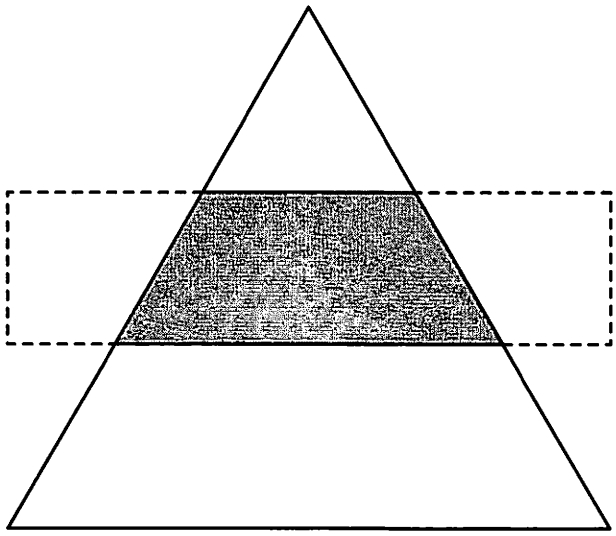


Figure 26-1. The Clipping Path

The triangle shape in Figure 26-1 has been defined within a path bracket and selected as the current clipping path. The filled box shape is drawn subsequently, and is therefore clipped to the triangle. The broken lines show the area of the box that has been clipped away.

Clipping paths are most useful when you want to use an irregular clipping boundary, or when the clipping boundary is itself an integral part of the picture. Both are true of the clipping path in Figure 26-1.

Any drawing that is clipped to the current clipping path is treated like an area-fill pattern. That is, the clipping path can be filled in alternate mode or in winding mode, except that a modified path can be filled in winding mode only. For a description of the area-fill modes, see Chapter 20, "Graphic Primitives and Their Attributes." Any point on the boundary of the path is considered to be within the path and is not clipped.

To terminate clipping to the current clipping path, you issue the GpiSetClipPath (GSSCLP) call with an identifier of '0'. This call deselects the current clipping path. The current clipping path is deselected automatically in some circumstances. For more information, see Appendix C, "Graphics Attributes."

The Viewing Window

The viewing window is a rectangular clipping boundary in model space, which you define using the GpiSetViewingLimits (GISVL) call. As input to this call you supply the model-space coordinates of the lower-left and upper-right corners of the viewing window. You define a viewing window when part of a model space, rather than the whole, is to be transformed to presentation-page space.

Any point on the boundary of the viewing window is considered to be within the window, and is not clipped. When a clipping path intersects a viewing window, that part of the path outside the viewing window is clipped away as it is drawn. If you allow the viewing window to default, the whole model space is transformed. Figure 26-2 on page 26-3 shows how the viewing window outlines a part of model space.

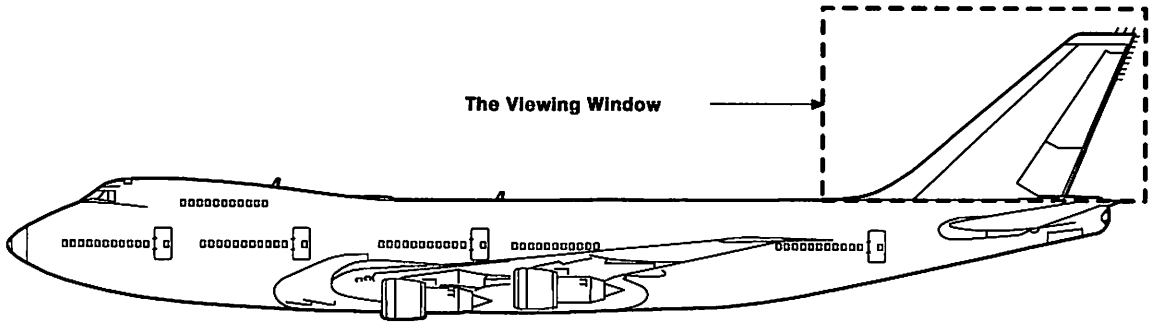


Figure 26-2. The Viewing Window. This example shows how the presentation page in Figure 25-3 on page 25-5 is constructed. The viewing window outlines the tail of the aircraft, which is scaled and translated when drawn in presentation-page space. The rest of the aircraft is clipped away during the drawing process.

The Graphics Field

The graphics field is a rectangular clipping boundary that lets you print or display a part, rather than the whole, of the picture assembled in presentation-page space. To specify the size of the graphics field in presentation-page coordinates, you use the GpiSetGraphicsField (GIFLD) call. By default, there is no graphics-field clipping. If you do specify a graphics field, however, any point on its boundary is considered to be within the graphics field, and is not clipped. Anything transformed to the presentation page that falls outside the graphics field, but within the extent of the presentation page, is discarded. Consequently, any part of a clipping path or viewing window that does not fall within the graphics field is clipped away as it is drawn. Figure 26-3 shows how a graphics field could be defined for the presentation page in Figure 25-3 on page 25-5.

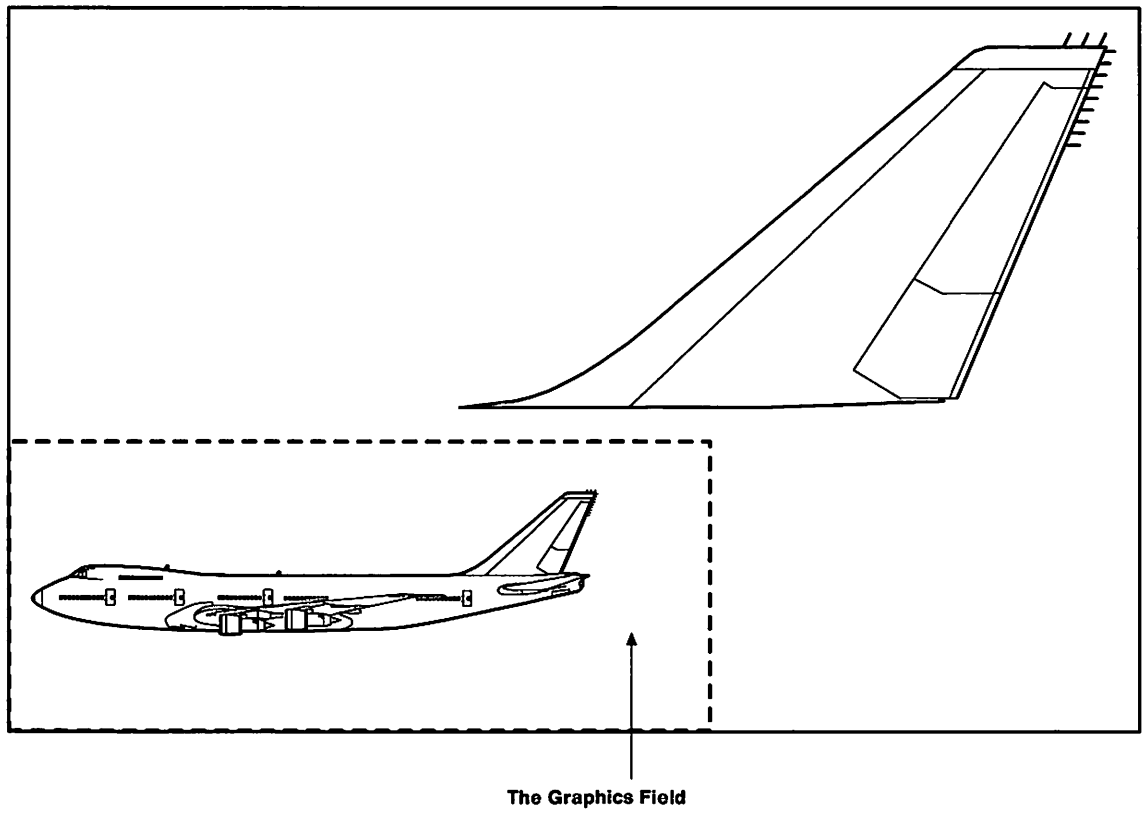


Figure 26-3. The Graphics Field. The broken line shows an arbitrary graphics field that is smaller than the presentation page. The aircraft tail is outside the graphics field and is clipped away as it is drawn.

The picture assembled in the graphics field is the picture that is displayed or printed. If you do not define a graphics field, the picture assembled in the presentation page is the picture that is displayed or printed.

Regions

Regions are similar to paths, in that they exist independently of their clipping function. You can create a region, update the region definition, draw the shape defined by the region and fill it with the current area-fill pattern, and delete the region definition, without using it as a clipping boundary. To use a region for clipping purposes, you have to select it as the current clipping region.

This section describes how regions are created and drawn, and how to use them as clipping boundaries.

Defining a Region

A region is made up of one or more overlapping or disjoint rectangles. To define a region, you use the GpiCreateRegion (GIREG) call. On input to GpiCreateRegion, you supply the coordinates of each rectangle that contributes to the region. Region coordinates must be in the range −32 767 through +32 765. The new region results from the union of the rectangles, as shown in Figure 26-4.

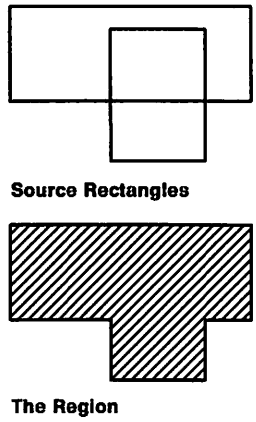


Figure 26-4. Defining a Region. This region comprises two overlapping rectangles. If the rectangles have no common elements, the region is defined as separate rectangles.

Regions are device-dependent and are defined in device coordinates. Each region is created for the device that is currently associated with the presentation space. The output from GpiCreateRegion (GIREG) is the *region handle*, which you use to identify the region in subsequent operations.

Updating a Region Definition

You can change the definition of an existing region using the GpiSetRegion (GIREGN) call. On input to GpiSetRegion, you supply the handle of the region to be updated, plus the dimensions of the new rectangles that replace the existing region definition. The updated region results from the union of the new rectangles, and the original region definition is discarded.

Combining Regions

To combine two existing regions to update the definition of a third region, you use the GpiCombineRegion (GSCREG) call. On input you supply the region handles of the two regions to be combined (the source regions), and the handle of the third region (the destination region). The destination region can be one of the two source regions, in which case that region is replaced by the new region. Any regions you combine must have been created for the same type of device.

The ways in which you can combine two regions to produce a third region are:

CRGN_OR
which is the union of the two source regions.

CRGN_XOR
which is the symmetric difference of the two source regions.

CRGN_COPY
which is source-region one only. Source-region two is discarded.

CRGN_AND
which is the intersection of the two source regions.

CRGN_DIFF
which is source-region one minus source-region two.

The effects of these different operations on overlapping regions are shown in Figure 26-5. Their effects on disjoint regions are shown in Figure 26-6.

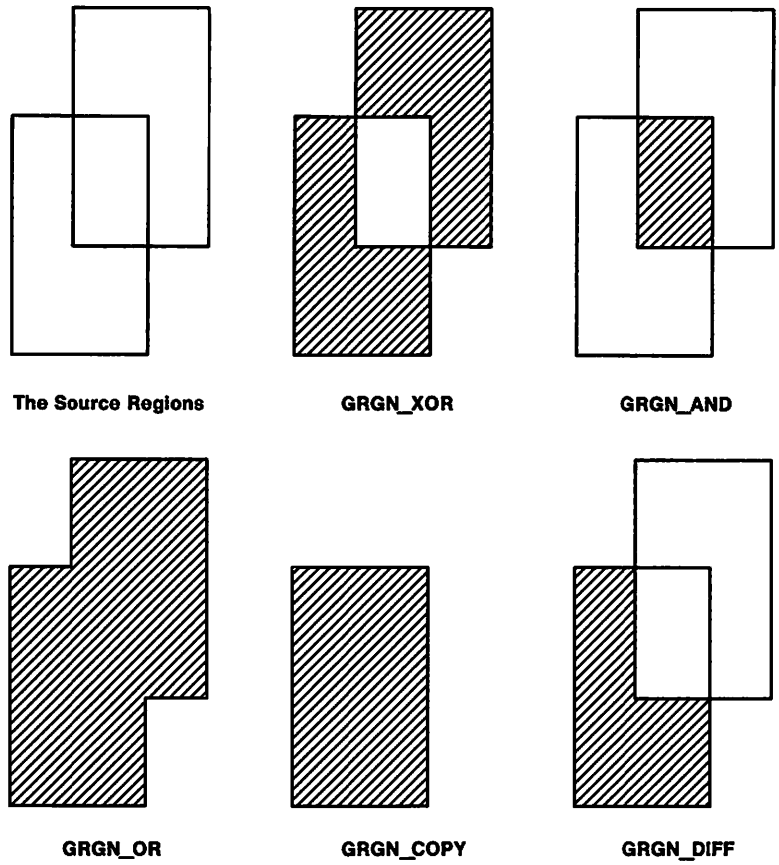


Figure 26-5. Combining Overlapping Regions. The source regions are two overlapping rectangles. The regions that result from the various combination operations are shown as shaded areas.

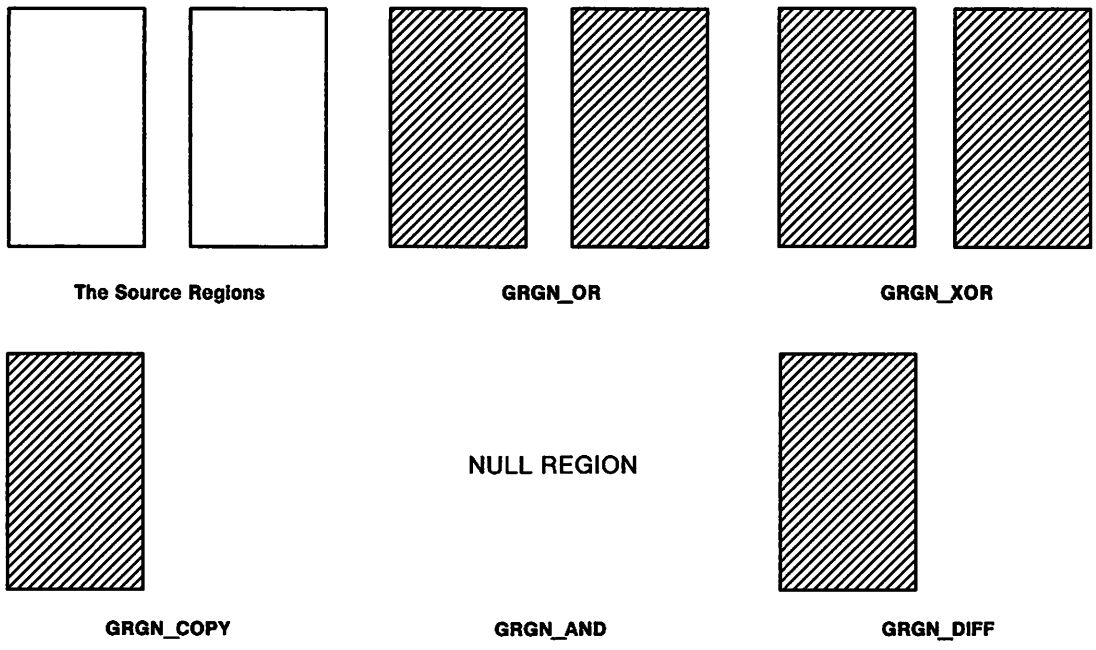


Figure 26-6. Combining Disjoint Regions. The two source regions do not overlap. The regions that result from the various combination operations are shown as shaded areas. The region resulting from a CRGN_AND combination of disjoint regions is a null region.

Output from the GpiCombineRegion (GSCREG) call tells you whether the resulting region is a null region, a rectangular region, or a complex region. A complex region is any region whose coordinates do not define a single rectangle.

Region-Manipulation Functions

The Presentation Manager supplies calls to let you retrieve information about an existing region or regions, move a region, fill a region with the current area-fill pattern, or delete a region.

- The GpiEqualRegion (GSEREG) call tells you whether two regions of the same device class are identical in shape.
- The GpiOffsetRegion (GIOREG) call moves a region by a specific number of device coordinates.
- The GpiPtInRegion (GIPIR) call tells you whether a particular point, expressed in device coordinates, lies within a named region.
- The GpiRectInRegion (GIRIR) call tells you whether any part of a rectangle (whose coordinates you supply) lies within a named region.
- The GpiQueryRegionBox (GIQRB) call returns the coordinates of the smallest rectangle that encloses a named region. Output from this call tells you whether the region itself is rectangular, null, or complex.
- The GpiQueryRegionRects (GIQRR) call returns the coordinates of the individual rectangles that, when combined using a CRGN_OR operation, make up a named region.
- To draw a region to the screen and fill it with the current area-fill pattern, you use the GpiPaintRegion call. The GpiPaintRegion (GSPREG) call does not cause graphics orders to be added to the current segment. Therefore, in retain or draw-and-retain mode, you are advised to use paths or areas rather than regions to ensure the desired effect.
- The GpiDestroyRegion (GSDREG) call deletes the definition of a named region.

Using the Region as a Clipping Boundary

When the screen contents are altered (for example, when a window is sized), you have to be able to repair or *heal* the part of the screen image affected by the change. To improve the performance of the drawing operation, you can restrict the redrawing and repair work to the affected parts of the screen.

You can do this by issuing the WinQueryUpdateRegion (WIQURG) call, which returns the coordinates of the window's update region. You define a clipping region using the dimensions of the update region, and issue an appropriate GPI drawing request, such as GpiDrawChain (GSDCHN), to redraw the screen contents. Any drawing that would occur outside the clipping region is discarded according to the rules described on page 26-1. Only those graphics within the clipping region are redrawn.

Selecting the Current Clipping Region

Regions are not automatically available for clipping. To select an existing region as the current clipping region, you use the GpiSetClipRegion (GSSCLR) call. There can be only one clipping region at a time in the presentation space. To terminate clipping to the current clipping region, you deselect the clipping region, using GpiSetClipRegion (GSSCLR) with a null region handle. A deselected clipping region retains the effects of any changes made to it while it was a clipping region, and it can be reselected.

You do not have to deselect the current clipping region before selecting another. Each selected clipping region automatically replaces the one before it. If there is already a clipping region when you issue GpiSetClipRegion (GSSCLR), the existing clipping region reverts to being a normal region, and its handle is returned to you.

Although a clipping region can comprise a number of overlapping or disjoint rectangles, do not use such regions for any purpose other than healing the screen because performance may be degraded. When you want to use a nonrectangular clipping boundary for any purpose other than healing the screen, you should use the clipping-path function.

The GpiSetClipRegion (GSSCLR) call does not cause graphics orders to be added to the current segment. Therefore, if the current drawing mode is retain or draw-and-retain, the effects of the clipping region are not guaranteed to be retained.

When you have selected the current clipping region, none of the region-related GPI calls discussed so far can be used for that region. Any of the following calls can be used to get information about, or to redefine, the current clipping region.

- GpiQueryClipBox
- GpiIntersectClipRectangle
- GpiExcludeClipRectangle
- GpiOffsetClipRegion
- GpiPtVisible
- GpiRectVisible.

All of these calls work in world coordinates rather than device coordinates, and are therefore subject to current transformations. The calls are described below.

You can request the dimensions of the smallest rectangle that encloses all current clipping boundaries by issuing the GpiQueryClipBox (GIQCLB) call. Boundaries included in this calculation are:

- Current clipping path
- Current viewing window
- Current graphics field
- Current clipping region
- Visible region of the window.

The GpiIntersectClipRectangle (GIICR) call redefines the current clipping region to be the intersection of the existing clipping region with the rectangle whose dimensions you supply on this call. This has the same effect as a CRGN_AND operation in the GpiCombineRegion (GSCREG) call. You can also redefine the current clipping region using the GpiExcludeClipRectangle (GIECR) call. This call excludes a specified rectangle from the current region, and has the same effect as a CRGN_DIFF operation on the GpiCombineRegion (GSCREG) call.

The current clipping region can be moved from its current position using the GpiOffsetClipRegion (GIOCR) call.

The GpiPtVisible (GIPVIS) call tells you whether a point, expressed in world coordinates, is visible on the screen. A point is visible if it is within all current clipping boundaries and is in the visible region of the window. The GpiRectVisible (GIRVIS) call tells you whether any part of a rectangle, whose dimensions you supply in world coordinates, is visible on the screen. The rectangle is visible if it intersects both the visible region of the window and all current clipping boundaries.

Redrawing Nondynamic Graphics

An interactive graphics application usually allows changes to be made to the displayed picture. For example, an object can be moved or sized, and you can plan for this by defining particular segments as dynamic. If dynamic segments are inappropriate (when you are using nonretained graphics, for example), you can repair the picture using a clipping region.

You may draw a segment chain comprising a circle and a hexagon, for example. If the circle segment is moved to another screen position by operator use of an input device, you have to repair its original location, and redraw it in its new location. To do this, you:

1. Determine the size of the smallest rectangle that contains the circle in its current position using a process known as *boundary determination.* Boundary determination is described on page 26-8.
2. Switch off the DCTL_DISPLAY flag of GpiSetDrawControl (GSDC), apply a translation transformation to the circle, and redraw it in its new position.
3. Determine the size of the smallest rectangle that contains the circle in its new position using boundary determination.
4. Use the GpiConvert (GICONV) call to convert the model-space coordinates provided by the boundary-determination process to device-space coordinates.
5. Use the device-space coordinates of the two rectangles to create a region, and select it as the current clipping region.
6. Switch on the DCTL_DISPLAY flag of GpiSetDrawControl.
7. Issue GpiErase (GSERSE) (or set the erase-before-draw control) to erase the current contents of the clipping region.

8. Redraw the segment chain, with the circle in its new position. Any part of the chain within the clipping region is redrawn. That part of the hexagon that is outside the clipping region is unaffected by the change, and does not have to be redrawn.

Figure 26-7 shows this sequence of events.

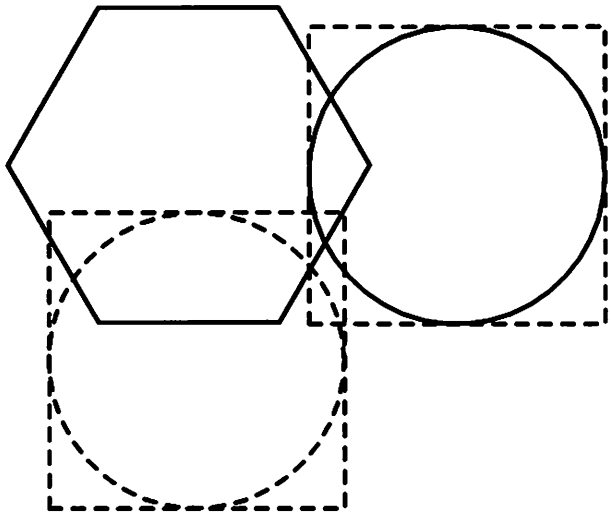


Figure 26-7. Defining a Clipping Region

The broken circle in Figure 26-7 shows the position to which the circle is to be moved. The two bounding rectangles overlap, and produce a complex region. If the circle were to be moved much further away from its start position, the region would comprise two disjoint rectangles. By default, the clipping region is the same size as the drawing surface.

Boundary Determination

Boundary determination is an operation to compute the size of the smallest rectangle that encloses a graphics object on the screen. The purpose of boundary determination is to allow you to heal only the affected parts of the screen when a graphics object is moved, for example, or when a graphics object is changed in some other way.

Dynamic segments are not included in boundary-determination operations.

Boundary determination can be performed on both retained and nonretained graphics. In both instances, you ask for boundary data to be calculated by setting the *boundary data flag* (DCTL_BOUNDARY) on the GpiSetDrawControl (GSDC) call. If you do not set this flag (for example, if you do not want to collect boundary data unnecessarily) and later find that you need boundary data for a particular object, you can:

1. Switch on the boundary-data flag, and switch off the display flag, using GpiSetDrawControl.
2. Redraw the object in its current location. Boundary data is collected, but the object is unaltered.

If you are drawing retained graphics, each drawing request (GpiDrawSegment, GpiDrawFrom, and GpiDrawChain) causes the boundary data resulting from the drawing to be made available. The application must request this data explicitly by issuing the GpiQueryBoundaryData (GIQBD) call after each drawing request for which it wants to examine boundary data. Boundary data is returned to the application in model space coordinates. The boundary data is reset before each drawing operation, so there is no risk of accumulating data from separate operations.

If you are drawing nonretained graphics, boundary data is accumulated for each GpiPutData (GSPUT) call and for each individual primitive-drawing call. The application can request the accumulated boundary data at any time by issuing the GpiQueryBoundaryData (GIQBD) call. Data continues to accumulate unless you issue the GpiResetBoundaryData (GSRBD) call. It is not reset automatically.

The boundary data returned to you is in the form of four model-space coordinates, which are the lowest x and y positions and the highest x and y positions of the bounding rectangle.

Example Code: Boundary Determination

This example, which is taken from the Graphic1 sample program, shows how to retrieve the boundary data for a single segment.

```
void SetRect(lprcSegRect)
PRECTL lprcSegRect;
{

  GpiResetBoundaryData(hpsClient);
  GpiSetDrawControl(hpsClient, DCTL_DISPLAY, DCTL_OFF);
  GpiSetDrawControl(hpsClient, DCTL_BOUNDARY, DCTL_ON);

  /*********************************************************************/
  /* The boundary data is reset to ensure that there is no residual    */
  /* data from earlier operations. The display drawing control is      */
  /* switched off, and the boundary-data control is switched on.       */
  /*********************************************************************/

  if (!fChained){
     GpiCallSegmentMatrix(hpsClient,
                          ulEditSeg,
                         (ULONG)9,
                          &amatlfSegXform[ulRootSeg],
                          TRANSFORM_REPLACE);
    } else {
      GpiDrawSegment(hpsClient, ulRootSeg);
    } /* endif */

  /*********************************************************************/
  /* The segment has to be redrawn to accumulate its boundary data. If */
  /* the segment is called from another segment, it is redrawn using   */
  /* GpiCallSegmentMatrix. Otherwise, GpiDrawSegment is used.          */
  /*********************************************************************/

  GpiSetDrawControl(hpsClient, DCTL_DISPLAY, DCTL_ON);
  GpiSetDrawControl(hpsClient, DCTL_BOUNDARY, DCTL_OFF);
  GpiQueryBoundaryData(hpsClient, (PRECTL)lprcSegRect);
  GpiConvert(hpsClient, CVTC_MODEL, CVTC_PAGE, (LONG)2, (PPOINTL)lprcSegRect);
  lprcSegRect->yTop++;
  lprcSegRect->xRight++;

  /*********************************************************************/
  /* When the segment has been drawn, the drawing controls are reset,  */
  /* and the boundary data is retrieved. It is then converted to       */
  /* presentation-page coordinates.                                    */
  /*********************************************************************/

  return;

}
```

Chapter 27. Saving and Restoring Graphics

The pictures that are created whenever a graphics application is executed are lost when the application finishes. For example, if your application creates a chain of retained segments to draw a picture of a house, that picture is created each time the application is run, and is lost when the application finishes. However, there are circumstances in which you might want to keep a picture beyond a single execution of an application. The Presentation Manager supports long-term retention of graphic data so that you can save pictures for use by any Presentation Manager application. The application can be:

- The application that created the graphics
- A different application on the same work station
- A different application on another work station.

You can also send pictures to non-Presentation Manager applications that support the MO-DCA (Mixed Object Document Content Architecture) interchange standard.

Data that is to be exchanged among Presentation Manager applications, or sent to non-Presentation Manager applications that support MO-DCA, is transferred in a *metafile*. A metafile can contain data generated from GPI calls only. Any nongraphical data included in a metafile is ignored.

Presentation Manager applications can accept graphic data from non-Presentation Manager applications that do not support MO-DCA format if the data is held in a *PIF* (Picture Interchange Format) file.[1] The use of PIF files is described on page 27-7.

Metafiles

A *metafile* is a file in which graphics are stored. The file is application-created, and it contains the graphics orders generated from those GPI calls that are valid in a metafile. Metafiled graphics can be reused by the application that created them. They can also be made available to other applications at the same, or at a different, workstation. Metafiles have several uses. For example:

- If your organization has common graphical resources, such as a company logo or a library of shapes that are generated by an application, those resources can be stored in a metafile and made available to other applications. This avoids the overhead of recreating the resources each time they are needed.
- As the metafile is a device-independent format, it is useful for transferring pictures that are to be printed (possibly from another node in the network) when you have no way of knowing the type of printer to which the picture is to be sent.
- Metafiles can be both created and used in draw mode by a single application. Therefore, an application that is not retaining segments in segment store (when a micro presentation space is being used, for example), can retain graphics in a metafile. The metafile is not a permanent object and, unless it is explicitly saved, disappears when the application completes. It does remain available, however, while the owning application is running, regardless of the number of presentation spaces the application obtains or defines. Graphics retained in this way can be rapidly redisplayed simply by executing the metafile contents. Given the same starting conditions in each presentation space, you can produce the same effects each time the metafile contents are executed.

A metafile contains all GPI instructions that contribute to the final version of a picture. (A metafile can contain data generated from GPI calls only. Any nongraphical data included in a metafile is ignored.) If you make a picture available to another application, you usually want that application to be able to reproduce the picture exactly as you created it. You may even want the receiving application to be able to edit the picture. Because of this, it is not enough to record only the drawing instructions for a picture or set of shapes in the metafile. If your intentions are to be clear to the receiving application, anything that contributes to

[1] Systems that do not support MO-DCA include GCP (Graphics Control Program), GGXA, GGXC, and GDDM (Graphical Data Display Manager).

the appearance of the picture has to be recorded. That is, items such as attribute-setting requests, color tables, and logical font descriptions, in addition to anything that describes the structure of the picture, have to be included. In other words, the contents of the metafile are likely to be similar, or identical, to those of the graphics presentation space. The Presentation Manager ensures that the environmental detail of the presentation space is recorded automatically in the metafile.

This section begins with a description of how to create a metafile, and then describes how to save it, how to execute its contents, and how to make it available to other applications. As metafiles are so versatile, there are some restrictions on what a metafile can contain in specific circumstances. In general, these restrictions are related to the drawing mode in which the metafile is to be used. These are described in "Restrictions on Metafile Contents" on page 27-5. For information about the structure of a metafile, see the *Programming Reference: Volume 2*.

Creating a Metafile

The Presentation Manager considers the metafile to be another of its output devices or destinations, along with screen windows and printers. Like those other destinations, the metafile requires a device context. To create a metafile device context, use the DevOpenDC (WDOPEN) call and specify a device type of OD_METAFILE. When the metafile device context has been associated with a graphics presentation space, the metafile is ready to receive graphical data. Just as with any other output destination, whether the picture is sent directly to the metafile is controlled by the current drawing mode:

- In draw mode, graphics go directly to the metafile.
- In retain mode, graphics go to the segment store of the presentation space. They are not directed to the metafile until the application issues an appropriate GpiDraw request.
- In draw-and-retain mode, graphics go directly to the metafile, and they also go to the segment store of the presentation space.

The drawing mode can be changed at any time while the metafile device context remains open.

Anything that can be described as a resource of the presentation space (for example, the current logical color table or logical font definition) is copied automatically into the metafile. Similarly, environmental detail, such as the presentation-page format, is recorded. You do not have to do anything to make sure that this happens, even though some of these items will have been established *before* the presentation space is associated with the metafile device context. You do have to ensure that any attribute settings, segment-creation requests, and primitive-drawing requests that contribute to the picture are directed to the metafile after its device context has been associated with a presentation space.

While the metafile device context remains open, you can continue drawing to it. Most GPI functions are valid in metafiles. You should, however, note the following:

- When you create a metafile from a micro-PS, or play the metafile back through a micro-PS, the contents of the metafile are restricted to those GPI functions that are valid in a micro-PS.
- Query functions can be issued, with the usual restrictions, to a presentation space associated with a metafile device context.
- The picture recorded in the metafile is the picture defined within the presentation page. Any drawing beyond the range of the presentation page is not recorded.
- Chained and unchained segments invoked by any GpiDraw command are written to the metafile.
- Primitives outside segments are recorded automatically as zero segments. If the metafile is subsequently played in DM_RETAIN mode, all graphics are directed to the segment store.
- Chained dynamic segments cannot be recorded in a metafile. The GpiRemoveDynamics (GSRDYN) and GpiDrawDynamics (GSDDYN) calls cause an error condition to be raised when the presentation space is associated with a metafile device context. Unchained dynamic segments are recorded as zero segments. (To draw an unchained dynamic segment, use the GpiDrawSegment (GSDSEG) call.)
- If the metafile is being created on an asynchronous drawing thread and the thread is suspended by the GpiSetStopDraw (GSSDW) call, an unusable metafile results.

There are some additional restrictions on the metafile contents that vary depending on how the metafile is to be used. These are described in "Restrictions on Metafile Contents" on page 27-5.

When you have finished drawing in the metafile, you close it using the DevCloseDC (WDCLS) call. A closed metafile cannot be reopened. Therefore, additional drawing in the metafile is not possible. The output from the DevCloseDC call is the *metafile handle*. Any reference to the metafile after it has been closed requires the metafile handle to be supplied as an input value.

At this point (when you have the handle of a newly created metafile), you can:

- Save it so that it can be made available to other applications at different work stations. If you do not save a newly created metafile, it is lost at the end of the session.
- Play it into a graphics presentation space.
- Make it available to other applications at the same work station.
- Edit it.
- Copy it.
- Delete it.

Saving a Metafile

The GpiSaveMetaFile (GSMSAV) call saves a metafile in a disk file. On input to this call, you supply the metafile handle and the name of the disk file in which the metafile is to be saved. As the disk file is created by this call, an error condition is raised if a file with this name already exists. When you issue GpiSaveMetaFile (GSMSAV), the memory version of the metafile is deleted, and its handle is no longer valid.

Loading a Metafile

A metafile saved on disk has to be reloaded before it can be used. Any application with access to the disk file can load the metafile using GpiLoadMetaFile (GSMLOD). Input to this call is the name of the disk file in which the metafile has been stored. Output from the call is the metafile handle, which identifies the metafile while it is loaded.

Playing a Metafile into a Graphics Presentation Space

An application can execute the contents of metafile. This is known as *playing* the metafile into a graphics presentation space. What this actually means depends on the current drawing-mode parameter in the target presentation space:

- When the drawing-mode parameter is DM_DRAW, the metafile contents are executed, and the picture is directed to the current output device.
- When the drawing-mode parameter is DM_RETAIN, the metafile contents are retained in the application's segment store. The picture defined in the metafile is not directed to an output device until the application issues an appropriate GpiDraw call.
- When the drawing-mode parameter is DM_DRAWANDRETAIN, the metafile contents are both stored and executed.
- Unchained segments in the metafile are always retained, regardless of the current drawing-mode parameter.

When the contents of a metafile are retained in the segment store of the target presentation space, they can be manipulated by the application as if the application had created them. For example, the segments can be edited, drawn, correlated on, and deleted. If the metafile contains a segment chain and is played when the drawing mode parameter is DM_RETAIN or DM_DRAWANDRETAIN, the chain is added to the end of any chain that may already be in the segment store. If any segment in the metafile has the same nonzero name as a segment already in the segment store of the presentation space, an error condition is raised.

To play a metafile you use the GpiPlayMetaFile (GSLOAD) call, which must not be issued while there is an open segment. GpiPlayMetaFile has several input parameters, some of which are discussed below. For the complete syntax of this call, see the *Programming Reference: Volume 1*.

The GpiPlayMetaFile (GSLOAD) Call

Before you play a metafile into a presentation space, you must make sure that the presentation-page units of the target presentation space are the same as those with which the metafile was generated. For example, if the original presentation page

is defined in increments of 0.1 mm, the target presentation page must be defined in the same way.

If the presentation-page formats do not match, you have to change the format of the target presentation page. You can do this using either the GpiSetPS (GIPS) call (if you know what the metafile presentation-page units are) or the PMF_RESET option of GpiPlayMetaFile (GSLOAD) with the value RES_RESET. This value redefines the target presentation page so that it is the same size, and defined in the same units, as the original presentation page. You do not need to know what these are. It also:

- Establishes default transformations as for a newly created presentation space.
- Ensures that the physical size of the metafile picture is preserved if the presentation page is defined in metric units.
- Deletes everything from the presentation space. This is equivalent to specifying GRES_ALL on the GpiResetPS (GSRPS) call.
- Sets the default viewing transformation in the presentation space to the value specified in the metafile.

By default, no resetting of the presentation space is performed. When you want to reset some or all of the current values in the target presentation space, without redefining the presentation page, use the GpiResetPS (GSRPS) call.

The PMF_LOADTYPE option of GpiPlayMetaFile (GSLOAD) determines which transformations affect the metafile picture. When you specify LT_NOMODIFY, which is the default value:

- Any viewing transformations recorded in the metafile are ignored. The metafile contents are drawn in accordance with the current viewing transformation in the target presentation space. (Note that, if you also specify RES_RESET, the current viewing transformation is reset to its default value.)
- The picture is clipped to any graphics field defined in the target presentation space, and any graphics field recorded in the metafile is ignored.
- Changes to the default viewing transformation recorded in the metafile are ignored.

When you specify LT_ORIGINALVIEW:

- Viewing transformations recorded in the metafile are used in the target presentation space.
- If changes to the graphics field are recorded in the metafile, the graphics field in the target presentation space is updated, and the picture is clipped to that graphics field.
- Changes to the default viewing transformation recorded in the metafile are used to update the default viewing transformation in the target presentation space.

The PMF_RESET option RES_RESET changes values in the target presentation space. Therefore, the effect of the PMF_LOADTYPE option should always be considered in conjunction with the PMF_RESET option. For example, if you specify both RES_RESET and LT_NOMODIFY, the default viewing transformation in the target presentation space is updated with that from the metafile, even though LT_NOMODIFY means that such values should be ignored.

The PMF_SUPPRESS option of GpiPlayMetaFile (GSLOAD) lets you prevent execution of the metafile contents while you are resetting the presentation space. You can issue two GpiPlayMetaFile calls, the first with the value SUP_SUPPRESS, so that you can reset the target presentation space, and the second with the value SUP_NOSUPPRESS, so that you can load the metafile contents. You can also make changes to the presentation space (such as setting up color tables or logical fonts) between these two calls.

The PMF_LCIDS option of GpiPlayMetaFile controls whether logical resources referenced by lcids (that is, logical fonts and bit maps used as area-fill patterns) are loaded from the metafile. If you allow this option to default (LC_NOLOAD) such resources are not loaded from the metafile. LC_LOADDISC causes lcid-referenced resources to be loaded from the metafile. If the lcids of those resources are already in use in the target presentation space, the resources currently identified by those lcids are deleted and their lcids are freed before the metafile contents are loaded.

The output from the desc parameter of GpiPlayMetaFile is a natural-language description of the picture contents. For example, its value might be *A House*. This description is provided on input to the DevOpenDC (WDOPEN) call that

defines the metafile device context. The desc value could be displayed in a list box, for example, to enable the application user to select a picture.

Making a Metafile Available to Other Applications at the Same Workstation

To make a metafile that you have created or loaded available to other applications at the same workstation, you place the metafile handle on the clipboard. (When a resource is passed to the clipboard, it becomes the property of the system, and is not deleted when the owning application terminates.) Any application that is able to access the clipboard can retrieve the handle, and supply it on input to the GpiPlayMetaFile (GSLOAD) call.

For information about using the clipboard for data transfer, see Chapter 28, "User-Driven Data Exchange – The Clipboard."

Editing Metafiles

If you want to access the metafile contents (to edit them, for example), you can copy the metafile into application storage using the GpiQueryMetaFileBits (GSMGD) call. To find out the size of the metafile, you use the GpiQueryMetaFileLength (GSQMFL) call. You do not have to move the whole metafile into application storage. If you find that the file is too large to retrieve in one go, you can load it piece by piece into application storage. The data can be written back to the metafile using the GpiSetMetaFileBits (GSMFB) call.

If you want to save an edited version of the file in a disk file, it cannot be the same file (if any) from which the metafile was loaded. GpiSaveMetaFile (GSMSAV) raises an error condition if the disk file already exists.

Copying Metafiles

A metafile in memory (either loaded from a disk file or newly created) can be copied. To take a copy of a metafile in memory, use the GpiCopyMetaFile (GSMCPY) call. The only input value to this call is the handle of the source metafile. The Presentation Manager takes a copy of the metafile, and returns to your application a different handle, which you use to identify the new metafile.

Deleting Metafiles

To delete the memory version of a metafile, you use the GpiDeleteMetaFile (GSMDEL) call.

Restrictions on Metafile Contents

The restrictions on what can be included in a metafile are mainly dependent on the drawing mode in which the metafile is to be played. If the metafile is to be played when the current drawing-mode parameter is DM_DRAW (and in no other drawing mode), the restrictions listed in "Draw-Mode Restrictions" apply. If the metafile is to be played when the current drawing-mode parameter is DM_RETAIN or DM_DRAWANDRETAIN, or if the file is to conform to SAA, then the restrictions listed in "Creating Metafiles for Interchange" on page 27-6 apply.

Draw-Mode Restrictions

These restrictions on the contents of the metafile apply if the metafile is to be played when the current drawing mode parameter is DM_DRAW:

- If the DCTL_DISPLAY flag of GpiSetDrawControl (GSDC) is off when you are sending graphics to a metafile, the graphics will not be visible when the metafile is played. Their effects on the current position and on current attribute settings are recorded.
- Region functions are not recorded in a metafile. The effects of the following calls, however, are recorded:
 - GpiSetClipRegion
 - GpiIntersectClipRegion
 - GpiExcludeClipRegion
 - GpiOffsetClipRegion
 - GpiPaintRegion.

 When the metafile is played into a presentation space, temporary regions are created. When GpiPlayMetaFile completes, these temporary regions are automatically destroyed and the clipping region that was current, before the metafile was played, is restored.
- In general, escape functions identified by reserved escape codes (that is, escape codes in the range 0 through 32 767) are recorded in a metafile. However, the DevEscape functions DEVESC_QUERYESCSUPPORT (escape code 0) and DEVESC_GETSCALINGFACTOR (escape code 1) are not stored in the metafile.

For more information, see the description of the DevEscape (WDESC) call in the *Programming Reference: Volume 1*.

- The effect of any GpiErase (GSERSE) call, including close-segment processing, is recorded in a metafile.
- The following calls can produce unexpected effects if the metafile contents are displayed on a different device from the one they were created for:
 - GpiBitBlt (GIBB)
 - GpiSetPel (GIPEL)
 - GpiSetClipRegion (GSSCLR)
 - GpiOffsetClipRegion (GIOCR)
 - GpiPaintRegion (GSPREG)
 - GpiIntersectClipRectangle (GIICR)
 - GpiExcludeClipRectangle (GIECR).

 This is because the pel resolutions of the devices may differ. Raster operations (for example, GpiBitBlt) do not work on plotters.
- You can associate the metafile device context with a different presentation space while creating the metafile, but the new presentation space must have the same format as the original presentation space.

Creating Metafiles for Interchange

These restrictions on the contents of the metafile apply if the metafile is to be played when the current drawing-mode parameter is DM_DRAWANDRETAIN or DM_RETAIN, or if the metafile is to conform to SAA:

- Before you issue the first drawing instruction to a graphics presentation space that has been associated with a metafile device context, you must establish (or default) the following:
 - The graphics field (use GpiSetGraphicsField (GIFLD))
 - The color table (use GpiCreateLogColorTable (GSCLCT))
 - The code page for the default character set (use GpiSetCp (GSCPG))
 - The default viewing transformation (use GpiSetDefaultViewMatrix (GISDVM))
 - The settings of the drawing controls (use GpiSetDrawControl (GSDC); DCTL_DISPLAY must be set on)
 - The default values of attributes (use GpiSetDefAttrs (GIDATR))
 - The viewing limits (use GpiSetDefViewingLimits (GISDVL))
 - The primitive tag (use GpiSetDefTag (GSTAG))
 - The arc parameters (use GpiSetDefArcParams (GIDAP)).

 None of these values can be reset while the metafile device context remains open.

 Note that you must not specify a graphics field if the metafile is to conform to SAA. If a graphics field is specified, you can play the metafile in DM_RETAIN or DM_DRAWANDRETAIN mode, but the graphics field must be specified before the first drawing instruction is issued to the metafile. The effect of the graphics field in the target presentation space is controlled by the PMF_LOADTYPE option of GpiPlayMetafile.
- You can define logical fonts and identify bit maps to be used as area-fill patterns at any time. You must not, however, issue the GpiDeleteSetId (GSRSS) call against these resources after they have been established. Once assigned, therefore, lcids cannot be reused.
- The size of the logical-color-table data must not exceed 31KB.
- You must not use clipping regions. Therefore, none of the following calls is supported:
 - GpiSetClipRegion (GSSCLR)
 - GpiExcludeClipRectangle (GIECR)
 - GpiIntersectClipRectangle (GIICR)
 - GpiOffsetClipRegion (GIOCR).
- You must not reassociate the metafile device context.
- When you use a bit map as the source of a GpiWCBitBlt operation or as an area-fill pattern, it must not be modified.
- Only the following foreground mixes can be used:
 - FM_DEFAULT
 - FM_OR
 - FM_OVERPAINT
 - FM_LEAVEALONE.

- Only the following background mixes can be used:
 - BM_DEFAULT
 - BM_OVERPAINT
 - BM_LEAVEALONE.
- The following calls are not supported:
 - GpiBitBlt (GIBB)
 - GpiSetPel (GIPEL)
 - GpiSetPS (GIPS)
 - GpiResetPS (GSRPS)
 - GpiErase (GSERSE)
 - GpiPaintRegion (GSPREG)
 - DevEscape.

PIF Files

Presentation Manager applications can accept PIF (Picture Interchange Format) files from non-Presentation Manager applications and convert them to metafile format. Presentation Manager provides the **Convert Picture File** utility to do the conversion.

The Convert Picture File Utility

Convert Picture File is used to convert:

- PIF files to metafiles
- Symbol sets to fonts.

Convert Picture File has a user interface that you invoke from the Desktop Manager window. The interface enables you to select the type of conversion you want and the names of the files you want to convert. You can select up to 50 files. The new files are created in the same directory as the source files.

You can also invoke Convert Picture File from the OS/2 command prompt. Type the command:

```
PicIchg [[d:][path]filename.ext]
```

The file name is the name of the source file that you want converted. The file-name extension is the type of the source file. When you are converting a PIF file to a metafile, use a file-name extension of .PIF. When you are converting a symbol set to a font, use a file-name extension of .SMB. If you supply no parameters for the PicIchg command, the Convert Picture File window is displayed.

For more information on the Convert Picture File user interface, see *Getting Started.*

Invoking Convert Picture File from Your Application

You can convert PIF files and symbol sets from within your application using the PicIchg (PICONV) call. The syntax for this call is described in the *Programming Reference: Volume 1.* As parameters to the call, you must supply the names of the source and the target files, together with the type of conversion you require.

Converting from Picture Interchange Format to Metafile Format

The following PIF resources are not supported in metafiles:

- Symbol sets
- Pattern sets
- Line types.

Because of this, Convert Picture File makes these changes when converting from PIF to metafile format:

- Any reference to an external symbol set is changed to a reference to a font that has the same name as the symbol set. Vector symbol sets are converted to outline fonts.
- Any reference to an internal symbol set is changed to a reference to the default font character set (lcid 0).
- Any reference to an internal or external line type set is changed to a reference to the default line type (line type 0).
- Any reference to an internal or external pattern set is changed to a reference to the default pattern set (pattern set 0).

Part 5. Presentation Manager Applications – Data Transfer Between Running Applications

Chapter 28. User-Driven Data Exchange – The Clipboard

As already described, under OS/2 the user can see several windowed applications on the screen simultaneously. There are occasions when the user wants to perform the following operations on displayed data:

Cut remove from a window, leaving a gap in the source, and save for later use

Copy copy from a window, leaving the source intact, and save for later use

Paste paste the **cut** or **copied** data into the window of an application (the target).

The easiest way for the user to be able to do this is for the source application to provide a visually apparent and interactive user interface for marking the boundary of the data to be cut or copied. Similarly, the target application should provide the user with a similar means of pasting the data in the display area. OS/2 provides a facility called the *Clipboard* that the source and target applications can use as an intermediate holding area between the **cut**, **copy**, and **paste** operations. An advantage of the clipboard being an intermediate area is that the target application does not have to be running at the time that the data is cut or copied into the clipboard from the source application. An application that uses the clipboard can be the source of a cut or copy operation *and* the target of the paste operation. Applications that use the clipboard are, therefore, responsible for providing the user interface as source and target.

You are recommended to place the **cut**, **copy**, and **paste** commands in a menu displayed by your application. When the user selects **cut** or **copy**, your application can display a shadow box that the user can manipulate, either with the mouse or with the keyboard keys, to indicate to the application the rectangle of data that is to be cut or copied. For the user, the clipboard is a notional place where the data is placed during the cut or copy operation.

The user defines a point in the target window where the data is to be pasted. The paste operation can obscure or overwrite the data in the target window.

The clipboard is not intended for use as a general means of transferring data between applications; for this function, see Chapter 29, "Application-Driven Dynamic Data Exchange."

The user's view of the clipboard is shown in Figure 28-1 on page 28-2, where the user is copying some data between two windows on the screen. The windows can belong to the same application or to two different applications. The user may even want to copy the data to another part of the same window. In the example, one window belongs to a spreadsheet application, the other window belongs to a word processor application.

Application View of the Clipboard

Clipboard operations (**cut**, **copy**, and **paste**) are user-driven, but applications must support them if they are to fit into the integrated environment offered by the Presentation Manager and provide the services the user requires. (Text windowed applications are provided with access to the clipboard through the system menu.)

For the programmer, the clipboard is a real object (and not a notional place) that can contain one set of data at a time; the set of data that the user is copying. Data can be placed in and extracted from the clipboard, but only at the request of the user. At any time, only one application can have control of the clipboard.

The most important feature of the clipboard is that although it can only contain one set of data at a time, the same data can be in any of a wide range of formats *simultaneously*. For example, the data from the spreadsheet in Figure 28-1 on page 28-2 could be represented as text, or as a bit-map image of what is visible on the screen, or as part of a spreadsheet file in spreadsheet format. All are valid representations of the data as seen by the user and *all* can be placed into the clipboard simultaneously.

Applications should support as many formats of data as are practical and relevant to the data they display. This allows the data to be received by the broadest range of other applications, as the application does not know the target application for copied data. This is not known until the user does a paste operation with the data. Clearly, in the example, the word processor cannot be expected to support the spreadsheet format, as this has no rele-

vance to the work done by the word processor. However, a variety of common text formats should be supported.

An application should retrieve data from the clipboard in the data format that provides the highest level of information about the data. There is typically more context and meaning to more elaborate formats. For example, if a spreadsheet application is offered data in both text and spreadsheet formats, it should take the spreadsheet format, as this offers a greater amount of information about the data being copied. If the text format were accepted in this case, the linkages between the various pieces of text that are defined in the spreadsheet format will be lost. This would be particularly annoying for the end user if the data is being copied from one part of the spreadsheet window to another.

In the example shown in Figure 28-1 the spreadsheet application should provide data in both spreadsheet and text formats. In this case the word processor application can only accept the text format.

A standard set of data formats is defined by the Presentation Manager, such as text, graphics metafile, and bit map. However, applications can define their own private formats and also make them public by publishing them.

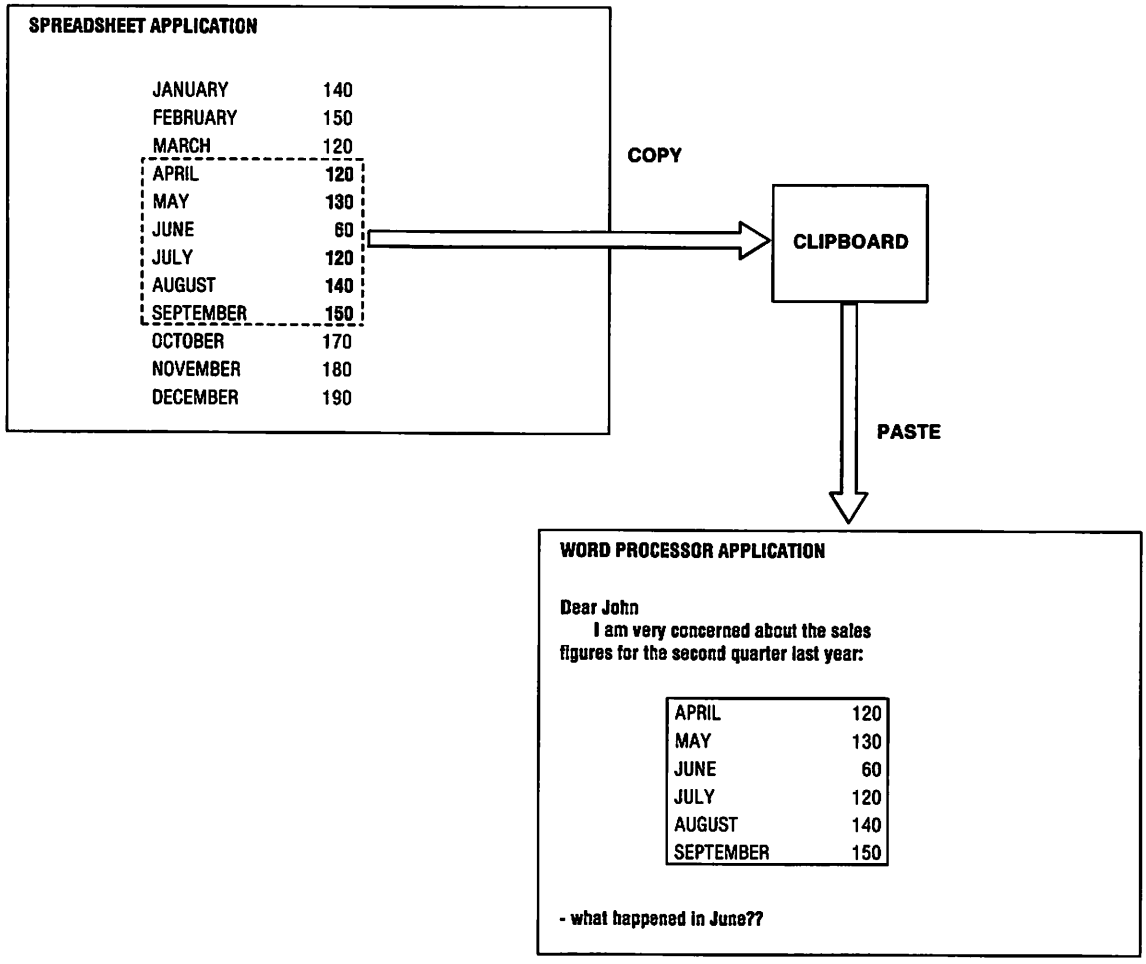


Figure 28-1. User's View of the Clipboard

The sequence of events involved in program use of the clipboard is as follows:

1. Put in data
 a. Open the clipboard
 b. Put the data into the clipboard
 c. Close the clipboard.
2. Retrieve data
 a. Open the clipboard
 b. Query data formats
 c. Select the "highest" form of data that the application can use
 d. Use the data
 e. Close the clipboard.

It is important *not* to close the clipboard before the data has been used.

Clipboard Owner

The clipboard is normally owned by the system. If you want an application to display the clipboard, rather than using it as a place to put data in or get data from, the application must issue a WinSetClipbrdOwner (WISCBO) call. OS/2 sends the clipboard owner appropriate messages when the clipboard contents are changed by another application.

Because the messages are sent, applications without window procedures cannot own the clipboard.

The CLIPBRD Sample PM Program

CLIPBRD displays a picture on which you can perform clipboard operations. The program illustrates the use of various clipboard, windowing, and graphics functions. For a description of the user interface of this program, see "The Clipboard C Sample – CLIPBRD" on page 2-7.

Program Description

Due to the size of the CLIPBRD sample, this chapter describes only those parts of it that illustrate techniques not described elsewhere in this book. CLIPBRD uses a standard Presentation Manager window. This is fully described in Chapter 5, "The Standard Window."

WNDPROC Window Procedure

The WNDPROC window procedure processes the message queue until it receives a WM_QUIT message. All actions are performed in response to messages received by this procedure.

Example Code: ClearRoutine

This routine is called when **Clear** is selected from the Edit pull-down.

The WinIsRectEmpty (WIREMP) call is used to test if a rectangle is defined. If one is, the EraseRect function deletes the contents of the rectangle from the memory bit map and repaints this part of the picture to show the change in the bit map. The WinSetRect (WISREC) call, with the parameters set to zero, then removes the rectangle itself.

```
VOID ClearRoutine()
{
     if (!WinIsRectEmpty(habMain, &tiStruct.rclTrack)){
         EraseRect();
         WinSetRect(habMain, &tiStruct.rclTrack, 0, 0, 0, 0);
     } /* endif */
}
```

Example Code: CopyRoutine

This routine is called when **Copy** is selected from the Edit pull-down.

The part of the picture defined by the rectangle is copied to a new bit map by a call to the GetBitmap routine. Provided that there is some data in the new bit map, the clipboard is opened using the WinOpenClipbrd (WIOPCB) call. The handle of the new bit map is passed to the clipboard using the WinSetClipbrdData (WISCBD) call to store the data in the clipboard. The clipboard is closed using the WinCloseClipbrd (WICCBD) call and the bounding rectangle is removed from the screen by a call to the RemoveRect function.

```
VOID CopyRoutine()
{
  HBITMAP bmapTempMem;

     bmapTempMem = GetBitmap();

     if (bmapTempMem != (HBITMAP)NULL){
         WinOpenClipbrd(habMain);
         WinSetClipbrdData(habMain, (ULONG)bmapTempMem, CF_BITMAP, 0L);
         WinCloseClipbrd(habMain);
         RemoveRect();
     } /* endif */
}
```

Example Code: CutRoutine

This routine is called when **Cut** is selected from the Edit pull-down.

CutRoutine copies data from a defining rectangle to the clipboard exactly as CopyRoutine does. In addition, the EraseRect function deletes, from the original bit map, the data defined by the rectangle, and repaints this part of the picture to show the change in the bit map. The WinSetRect (WISREC) call, with the parameters set to zero, then removes the rectangle itself.

```
VOID CutRoutine()
{
  HBITMAP bmapTempMem;

     bmapTempMem = GetBitmap();

     if (bmapTempMem != (HBITMAP)NULL){
         WinOpenClipbrd(habMain);
         WinSetClipbrdData(habMain, (ULONG)bmapTempMem, CF_BITMAP, 0L);
         WinCloseClipbrd(habMain);
         EraseRect();
         WinSetRect(habMain, &tiStruct.rclTrack, 0, 0, 0, 0);
     } /* endif */

}
```

Example Code: PasteRoutine

This routine is called when **Paste** is selected from the Edit pull-down.

The WinIsRectEmpty (WIREMP) call is used to test if a rectangle is defined. If one is, the clipboard is opened (by a WinOpenClipbrd (WIOPCB) call) and tested to see if it contains a bit map. If it does not, there is no data to paste, so the clipboard is closed using the WinCloseClipbrd (WICCBD) call and the WinSetRect (WISREC) call is used to remove the rectangle.

If there is data to paste, the size of the user-defined rectangle is calculated. Then the WinSetPointer (WMSPTR) call is used to change the pointer to an hour-glass symbol. This should be done whenever an operation may take some time. The clipboard bit map is copied to the memory bit map after various size adjustments, and then copied from there to the screen. Finally, the temporary resources are destroyed, the clipboard closed, the rectangle cleared, and the pointer restored to the usual arrow. It should be noted that the bit-map data is *not* deleted from the clipboard.

```
VOID PasteRoutine()
{
  HDC     hdcTempMem;
  HBITMAP bmapTempMem;
  HPS     hpsTempMem;
  BITMAPINFOHEADER bmpTemp;
  SIZEL   sizlWork;

     if (!WinIsRectEmpty(habMain, &tiStruct.rclTrack)){
         WinOpenClipbrd(habMain);
         bmapTempMem = (HBITMAP)WinQueryClipbrdData(habMain, CF_BITMAP);
         if (bmapTempMem != (HBITMAP)NULL){

  :
             /* Calculate size of rectangle */

  :
             WinSetPointer(HWND_DESKTOP,
                           WinQuerySysPointer(HWND_DESKTOP,
                                              SPTR_WAIT,
                                              FALSE)
                           );

  :
             /* Get clipboard data          */

  :
             WinSetPointer(HWND_DESKTOP,
                           WinQuerySysPointer(HWND_DESKTOP,
                                              SPTR_ARROW,
                                              FALSE)
                           );

             GpiSetBitmap(hpsTempMem, (HBITMAP)NULL);
             GpiAssociate(hpsTempMem, (HDC)NULL);
             GpiDestroyPS(hpsTempMem);
             DevCloseDC(hdcTempMem);
         } /* endif */
       WinCloseClipbrd(habMain);
       WinSetRect(habMain, &tiStruct.rclTrack, 0, 0, 0, 0);
```

```
    } /* endif */
}
```

Clipboard Function Call Summary

WinCloseClipbrd (WICCBD)
Closes the clipboard, so other applications can use it.

WinEmptyClipbrd (WIECB)
Empties the clipboard, removing and freeing all handles to data in the clipboard.

WinEnumClipbrdFmts (WIECF)
Enumerates the clipboard data formats available in the clipboard.

WinOpenClipbrd (WIOPCB)
Opens the clipboard for use.

WinQueryClipbrdOwner (WIQCBO)
Obtains the current clipboard owner window, if any.

WinQueryClipbrdData (WIQCBD)
Obtains a handle of current clipboard data in the specified format.

WinQueryClipbrdFmtInfo (WIQCBF)
Determines whether a format of data is present in the clipboard. If it is, information is provided about that format.

WinQueryClipbrdViewer (WIQCBV)
Returns the handle of the current clipboard viewer window (if any).

WinSetClipbrdData (WISCBD)
Puts data in the clipboard.

WinSetClipbrdOwner (WISCBO)
Sets the current clipboard owner window.

WinSetClipbrdViewer (WISCBV)
Sets the current clipboard viewer window.

Chapter 29. Application-Driven Dynamic Data Exchange

Chapter 28, "User-Driven Data Exchange – The Clipboard" describes how the user controls the exchange of data between applications, with the clipboard as an intermediate holding area. Clipboard data exchange directly involves the user, who must control when and where it happens.

You may want your applications to establish longer-term access to each other's data and to exchange data *without* involving the user. OS/2 provides a protocol called Dynamic Data Exchange (DDE), that makes this type of data exchange easier. Because the protocol is based on sending messages, this facility is not available to applications that do not have window procedures.

A simple example of DDE is a charting application that displays some data from a separate spreadsheet application. DDE enables the chart to be updated automatically when the user changes the data in the spreadsheet, as in Figure 29-1.

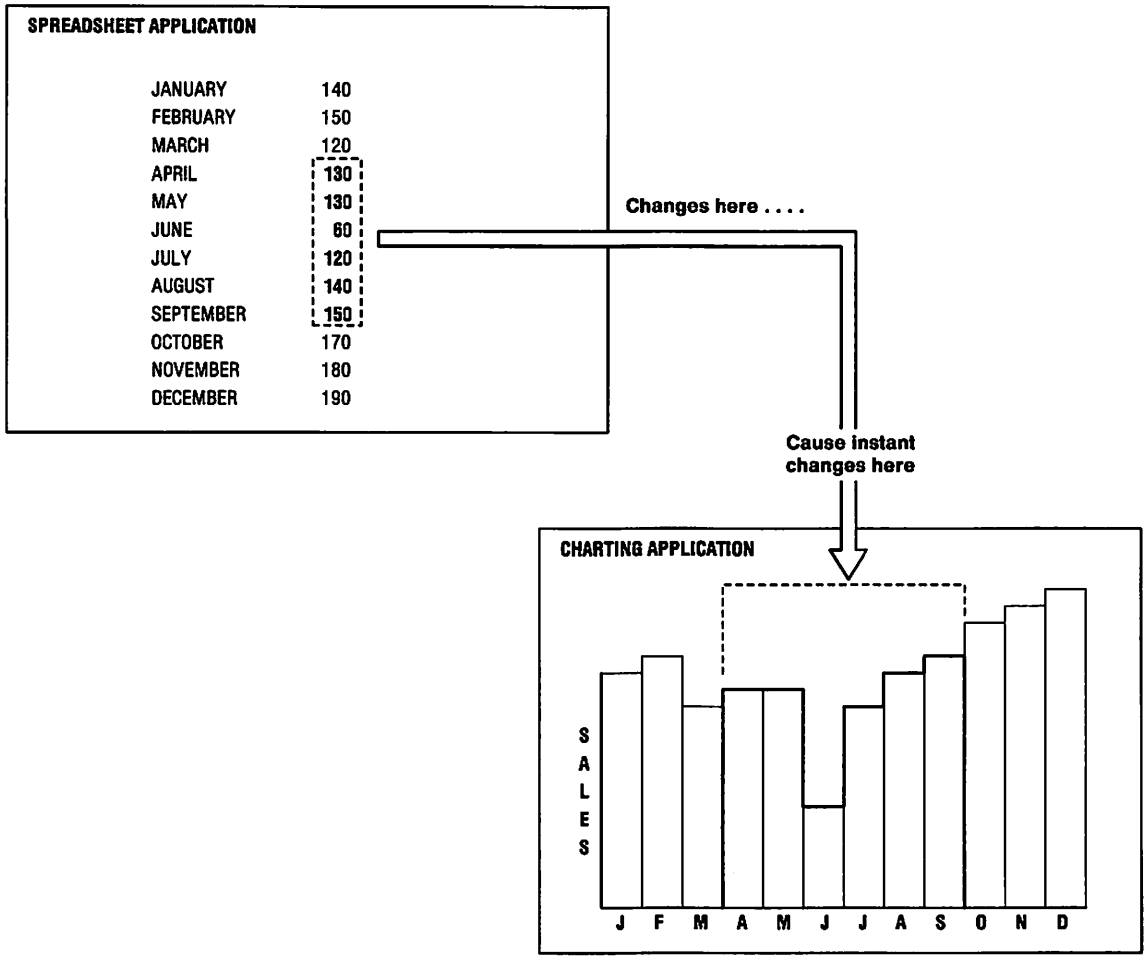


Figure 29-1. User's View of Dynamic Data Exchange

DDE is Message-Based

This section explains how the underlying Presentation Manager mechanisms allow easy implementation of the DDE Protocol.

Presentation Manager allows for very flexible inter-task communication. The primary means of communication is the passing of messages. Presentation Manager applications receive all of their input in the form of messages. All characters typed at the keyboard, all mouse activity, and all menu selection events are sent to the appropriate Presentation Manager application through messages. A Presentation Manager application is structured with a main loop which is always asking the system for messages and then processing them.

In Presentation Manager, applications may define private messages that have a unique meaning throughout the system. The DDE Protocol defines some new messages for communication between the applications that use it.

OS/2 provides for the sharing of data between applications. The DDE protocol uses shared memory as the means of transferring data from application to application. DDE defines a structure to be used for the contents of the shared memory objects.

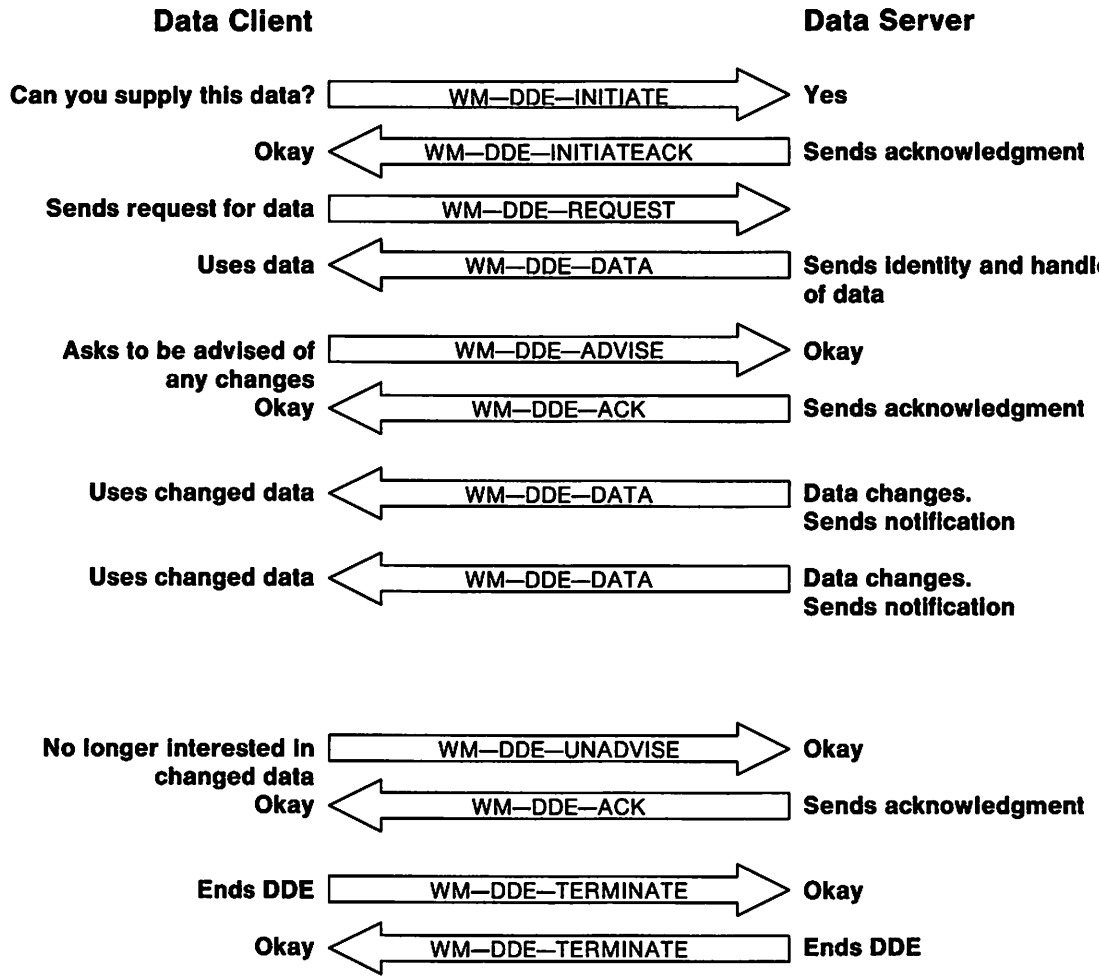


Figure 29-2. DDE Message Passing

Terminology and Basic Rules

DDE is a protocol that allows applications to exchange data on a real time basis. To perform such an exchange, the two participating applications first have to engage in a DDE conversation. The application that initiates the conversation is known as the client application, and the application responding to the client is known as the server application. A given application can be engaged in several conversations at the same time, and can act as a client application in some of them and as a server application in others.

A DDE conversation between two applications actually takes place between two windows, one for each of the participating applications. Applications open a window for each conversation they engage in. (Note that such windows are not typically visible.) A window is identified by its handle, hwnd. The window belonging to the server application is the server window; the window belonging to the client application is the client window.

DDE uses the three-level hierarchy (application, topic, item) to identify a unit of data uniquely. An item is a data object that can be passed in a DDE data exchange. For example, an item might be a single integer, a string, several paragraphs of text, or a bit map. A topic is a logical data context. For applications that operate on file-based documents, topics are usually file names; for other applications they are other application-specific strings. As each window represents an application's conversation on a single topic, the window identifier hwnd identifies both application and topic.

Several server windows can be associated with the same topic at the same time. Also, a given window can participate either as server or as client in several conversations on the same topic at the same time.

After a conversation has been started, the client interacts with the server by issuing transactions. When issuing a transaction, the client is asking the server to perform a given action. There are six types of transactions: REQUEST, ADVISE, UNADVISE, POKE, EXECUTE, and TERMINATE. These are permitted only within an INITIATEd conversation. Transactions in a DDE conversation are one-way: the client application is always the one that issues the transactions. If the server wants to issue a transaction to the client, the server is expected to initiate a new conversation for that purpose. The server becomes the client in this new conversation. (The only exception to the one-way rule is the TERMINATE transaction, which can be issued by either the client or the server.)

Presentation Manager defines DDE specific calls. The WM_DDE_INITIATE message and its corresponding WM_DDE_INITIATEACK are issued using WinDdeInitiate (WMDEIN), and WinDdeRespond (WMDERE), respectively, while all other messages are issued using the WinDdePostMsg (WMDEPO) call.

A complete sequence of DDE protocol between a client and a server application could involve, for example, the messages shown in Figure 29-2 on page 29-2.

Uses for Dynamic Data Exchange

The uses of DDE mentioned in this section are intended simply as a starting-off point. There are many possibilities for the use of DDE and here we can only suggest a few.

Potential uses of DDE abound in real-time data-acquisition applications. What follows is one such example, a DDE-based real-time portfolio tracking system. In this example, there are two hypothetical Presentation Manager applications cooperating. One is an application that receives stock-quote data. The other is a spreadsheet application. Both applications are built to use the DDE protocol. In the described DDE exchanges, the spreadsheet application is the client, that is, the application that initiates DDE transactions, and the stock-quote application is the server.

The example spreadsheet has the following layout:

	A	B	C	D
1	Stock	Shares	Price	Extension
2	XYZCO	1000	148	155000
3	SMITH	2000	26	80000
4	JONES	200	24	7000
5	BROWN	2000	93	76000
6				318000

Without DDE, this spreadsheet could be updated by using the Clipboard to copy numbers manually from the stock quote application screen display into the spreadsheet. This would require screen-sharing

or switching between applications, and would also require that the user pay attention to the price data, and undertake the data exchange when desired.

With DDE, this system could be much more automatic, providing the spreadsheet with the current values for multiple data items without intervention by the user. DDE would allow the user to set up a conversation between the server and client applications that would keep the spreadsheet informed of any changes in the value of the stocks that it has asked about.

A simple user interface might be for the spreadsheet to provide for external references in cell formulas; these could be designed to retrieve data from another application through DDE. An example syntax for such an external reference might be:

= Application | Topic ! Item

which in this spreadsheet might read:

= STOCKQUO | NYSE ! XYZCO

This indicates that the cell content is to be the value of the data item 'XYZCO' in the context 'NYSE' of the application 'STOCKQUO'.

When this connection was established, the cell value would always reflect the most current data available from the server. No per-transfer intervention on the user's part would be necessary. This would facilitate the analysis of real-time data in a timely manner.

The usefulness of DDE is not restricted to specialized real-time data acquisition applications. Productivity software in general can benefit significantly from the protocol. For example, suppose a report document is to be prepared monthly, using a graphics-and-text word processor. The report is to include graphs generated in a separate business graphics package. Without DDE, it would be necessary to go through a manual copy and paste process to include each month's new graphs into each month's report document. With DDE, the word processor can establish a permanent link to the charting application, so that any changes made by the user to the charting document are reflected in the word processing document, either automatically or on request. This makes the routine of document preparation much simpler for the user.

Inside DDE: the Workings of the Protocol

This section presents a more detailed view of the workings of the DDE protocol. You will be taken through the example of the stock-quote/spreadsheet interaction, illustrating the forwarding of stock quotes from the stock quote interface application to the spreadsheet. For the sake of simplicity, the example is limited to the exchange of quotes for a single stock, XYZCO.

The stock-quote DDE server application is started first. Typically, applications designed to operate as dedicated DDE servers have some user interface for initialization, and then run as icons at the bottom of the Presentation Manager screen. As part of the initialization process, the stock-quote DDE server application goes through whatever steps are necessary (entering passwords, tuning, testing, and such like) to insure that reception is good and that data can be provided to clients.

The spreadsheet is started next, and the stock-portfolio document is loaded. At this time, the spreadsheet broadcasts a WM_DDE_INITIATE message to all currently running applications.

The broadcast message is a request to initiate a conversation with the application called STOCKQUOTE on the topic called NYSE. An application receiving such a request can accept it by responding with a WM_DDE_INITIATEACK message, or can decline it by not responding. If no application accepted the request, the spreadsheet would assign an error value to the external reference, and its DDE activity would end. In the context of this example, it is assumed that the currently running stock-quote application accepts the request, and therefore that a DDE conversation is established between the two applications.

As part of its initialization, the spreadsheet application asks for a complete set of data by sending a WM_DDE_REQUEST message. The stock-quote application responds with a WM_DDE_DATA message providing the required data.

The spreadsheet can now use the newly established conversation to ask the stock-quote application for continuous updates on the specific data item it needs. To do so, it sends a WM_DDE_ADVISE message to the stock-quote application, indicating that it wants updates sent every time there is a new

value available for the data item named 'XYZCO', and that it wants the updates, say, in text format (CF_TEXT).

Upon receiving this message, the stock-quote application makes a note of the request in its database, and sends a WM_DDE_ACK. This is a simple example of the implementation of the protocol. A full description of the protocol's message sequences, the content of messages, and the makeup of shared memory objects is contained in "DDE Protocol Definition" and "DDE Implementation Rules."

DDE Protocol Definition

The Dynamic Data Exchange protocol is implemented using a set of ten messages whose purpose and parameters are described below. The descriptions in this section give only a limited indication of how to organize the conversation between applications; a more complete discussion of that topic starts at "DDE Implementation Rules."

Conventions Used in This Chapter

Parameter names bear prefixes indicating their type, as follows:

cfFormat
Registered clipboard format number

wName
Any other word-length parameter

fName
Flag bit

szName
Null-terminated string

pName.
Pointer to an object.

DDE Message Set

Each DDE message has two parameters. The first parameter, lParam1, carries the handle of the sender's window; it is the same in all instances, and so is not shown in the table below. In all but the initiation and termination messages, the second parameter, lParam2, carries a pointer to the DDE structure, DDESTRUCT.

Message Name	Argument in lParam2
WM_DDE_ACK	PDDESTRUCT
WM_DDE_ADVISE	PDDESTRUCT
WM_DDE_DATA	PDDESTRUCT
WM_DDE_EXECUTE	PDDESTRUCT
WM_DDE_INITIATE	PDDEINIT
WM_DDE_INITIATEACK	PDDEINIT
WM_DDE_POKE	PDDESTRUCT
WM_DDE_REQUEST	PDDESTRUCT
WM_DDE_TERMINATE	(reserved)
WM_DDE_UNADVISE	PDDESTRUCT

The DDE structure, DDESTRUCT is fully described in the *Control Program Programming Reference*. These messages are passed using the Presentation Manager WinDdeInitiate (WMDEIN), WinDdeRespond (WMDERE) and WinDdePostMsg (WMDEPO) calls. The handle hwnd of the addressee window appears as the first parameter of these calls, for example:

```
WinDdePostMsg(hwndTo,
              HwndFrom,
              wm,
              DDESTRUCT far*,
              cRetry)
```

These calls are fully described in the *Control Program Programming Reference*.

DDE Implementation Rules

This section is divided into five parts, each covering one of the five basic types of interchange supported by DDE:

1. Initiation of a conversation between two applications.
2. One-time data transfer between two applications.
3. Permanent data link between two applications.
4. Execution of commands in a remote application.

5. Termination of a conversation between two applications.

Each part provides a detailed description of the specific message protocols associated with the transactions it encompasses.

Initiation of a Conversation Between Two Applications

To initiate a DDE conversation, the client broadcasts an WM_DDE_INITIATE message as follows:

```
WinDdeInitiate(hwndClient,
               pszAppName,
               pszTopicName)
```

where:

hwndClient
window handle supplied by the calling application.

pszAppName and pszTopicName
pointers to null terminated strings.

pszAppName
points at a string containing the desired server application.

pszTopicName
points to a string containing the desired topic name.

The WinDdeInitiate (WMDEIN) call does not return to the client application until all receiving applications have, in sequence, returned control to Presentation Manager. szAppName and szTopicName can be left zero length by the client application. Server applications respond according to the rule below:

A recipient of a WM_DDE_INITIATE message responds only if it supports the topic requested in szTopicName. If szTopicName is left zero length, the recipient responds once for each topic it supports. If a recipient's name does not match the name passed in szAppName, or it does not support the topic requested, it does not respond.

To acknowledge the requested topic, server responds with:

```
WinDdeRespond(hwndClient,
              hwndServer,
              pszAppName,
              pszTopicName)
```

To acknowledge each supported topic, the server makes one response as shown above for each topic it is willing to support. This initiates a conversation on each topic; the client is expected to terminate all unwanted conversations. The responding server sets szAppName or szTopicName if they are received empty.

One-Time Data Transfer Between Two Applications

A client application can use DDE to obtain a data item from a server (REQUEST), or to submit a data item to a server (POKE).

REQUEST Transaction

The client sends the server a REQUEST message specifying the desired item and format, as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_REQUEST,
              DDESTRUCT far*,
              cRetry)
```

If the server has access to the desired item and can render it in the desired format, the server renders the item as in the data area below DDESTRUCT and sends the client a WM_DDE_DATA message as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_DATA,
              DDESTRUCT far*,
              cRetry)
```

If the server cannot satisfy the request, it sends the client a negative WM_DDE_ACK, as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_ACK,
              DDESTRUCT far*,
              cRetry)
```

A negative WM_DDE_ACK is sent by setting the DDE_FACK bit in DDESTRUCT to zero, indicating that the server application is unable to accept the request. Additionally the DDE_BUSY bit can be set to 1, identifying this as the reason for not accepting the request.

Upon receiving a WM_DDE_DATA message, the client processes the data item as appropriate. Then, if the DDE_FACKREQ bit for the item is 1, the client is expected to send the server a positive WM_DDE_ACK message as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_ACK,
              DDESTRUCT far*,
              cRetry)
```

Upon receiving a negative WM_DDE_ACK message, the client can ask for the same item again, specifying a different clipboard format. Typically a client will first ask for the richest format it can support, then step down if necessary through progressively simpler formats until it finds one the server can provide.

POKE Transaction

The client renders the item to be sent in the format of its choice, and sends the server a WM_DDE_POKE message as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_POKE,
              DDESTRUCT far*,
              cRetry)
```

If the server can accept the data item and the format in which it is rendered, the server processes the item as appropriate and sends a positive WM_DDE_ACK. If unable to process the item because of format or other reasons, the client sends a negative WM_DDE_ACK message. The positive and negative WM_DDE_ACK messages are of the form:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_ACK,
              DDESTRUCT far*,
              cRetry)
```

Permanent Data Link between Two Presentation Manager Applications

A client application can use DDE to establish a link to an item in a server application. When such a link is established, the server sends periodic updates of the linked item to the client (typically, whenever the data associated with the item in the server application changes). Thus, a permanent *data stream* is established between the two applications, and it remains in place until it is explicitly disconnected.

ADVISE Transaction (Establishing the Link)

The client sends the server an ADVISE message as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_ADVISE,
              DDESTRUCT far*,
              cRetry)
```

If the server has access to the requested item, and can render it in the desired format, the server notes the new link (remembering the flags specified in wStatus), and sends the client a positive WM_DDE_ACK. From then on, until the client issues a matching UNADVISE transaction, the server executes a WM_DDE_DATA data-update sequence (see below) every time the source data

associated with the item in the server application changes.

If the server is unable to service the request, it sends the client a negative WM_DDE_ACK.

WM_DDE_DATA Data Update Sequence

For links established with DDE_FNODATA=0, the client has requested that the data be sent each time it changes. In such instances, the server renders the new version of the item in the previously specified format, and sends the client a WM_DDE_DATA message as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_DATA,
              DDESTRUCT far*,
              cRetry)
```

The client processes the item as appropriate. If the DDE_FACKREQ bit for the item is 1, the client then sends the server a positive WM_DDE_ACK message.

For links established with DDE_FNODATA=1, the client has requested that only a notification, not the data itself, be sent each time the data changes. In such instances, when the source data changes the server does not render the new version of the item, but simply sends the client a WM_DDE_DATA message with a zero length data section as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_DATA,
              DDESTRUCT far*,
              cRetry)
```

At its discretion, the client can then request the latest version of the data by performing a regular WM_DDE_REQUEST transaction, or it can simply ignore the data change notice from the server. In either instance, if DDE_FACKREQ=1, the client is expected to send a positive WM_DDE_ACK to the server.

WM_DDE_UNADVISE Transaction (Terminating the Link)

If the client wants to terminate a single specific link, the client sends the server a WM_DDE_UNADVISE message as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_UNADVISE,
              DDESTRUCT far*,
              cRetry)
```

The server checks that the client currently has a link to the specified item in this conversation. If so, the server sends the client a positive WM_DDE_ACK message, and is then no longer responsible for sending updates on the item in this conversation. If the server has no such link, it sends a negative WM_DDE_ACK.

If the client wants to terminate all links for a particular conversation, the client sends the server a WM_DDE_UNADVISE message as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_UNADVISE,
              DDESTRUCT far*,
              cRetry)
```

with a zero-length szItemName.

The server checks that the conversation has at least one link currently established. If so, the server sends a positive WM_DDE_ACK message, and is then no longer responsible for sending any updates in the conversation. If the server has no links in the conversation, it sends a negative WM_DDE_ACK.

Execution of Commands in a Remote Application

A Presentation Manager application can use DDE to cause a specific command or series of commands to be executed in another application. Such remote executions are performed by using the WM_DDE_EXECUTE transaction.

The client sends the server a WM_DDE_EXECUTE message containing a handle to a command string, as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_EXECUTE,
              DDESTRUCT far*,
              cRetry)
```

The server tries to execute the specified string. If successful, the server sends the client a positive WM_DDE_ACK message; if unsuccessful, a negative WM_DDE_ACK message. These messages are of the following form:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_ACK,
              DDESTRUCT far*,
              cRetry)
```

See the *Control Program Programming Reference* for a complete definition of the flags to be set in wStatus.

Termination of a Conversation between Two Applications

Either client or server can issue a WM_DDE_TERMINATE message to terminate a conversation at any time, by following the procedure described below. Similarly, both client and server applications should be prepared to receive a WM_DDE_TERMINATE message at any time.

An application must WM_DDE_TERMINATE all of its conversations before shutting down.

The application wanting to terminate sends a WM_DDE_TERMINATE message, as follows:

```
WinDdePostMsg(hwndTo,
              hwndFrom,
              WM_DDE_TERMINATE,
              NULL,
              cRetry)
```

This constitutes a promise to send no further messages, and confers permission for the recipient to close its window (destroy its hwnd). The recipient is expected in all instances to send a WM_DDE_TERMINATE message promptly in response; it is not permissible to send a negative, busy, or positive WM_DDE_ACK.

If the original sender receives any other message before the WM_DDE_TERMINATE, no response must be sent, as the sender of that other message might already have destroyed the window that the response would be sent to.

Synchronization Rules

A window processing DDE requests from another window must process them strictly in the order in which the requests are received.

A window does not need to apply this first-in first-out rule between requests from different windows, that is, it can provide asynchronous support for multiple processes.

Example: If a window has the following requests in its queue:

1. Message from window x
2. Message from window y
3. Message from window x.

the window must process 1 before 3, but it does not need to process 2 before 3. It may decide that y is a lower priority client than x, and therefore follow the order 1, 3, 2.

If a server cannot process an incoming request because it is itself waiting for an external process, it must post a *busy* WM_DDE_ACK message to the client, to prevent deadlock. A busy WM_DDE_ACK may also be sent if the server expects for any reason to be unable to process an incoming request within a reasonable length of time.

Shared Memory Object Rules

DDE uses shared memory objects for all communications.

Shared memory objects need to be freed correctly to let the memory manager be effective.

The following rules specify the correct handling of memory objects:

Responsibilities of Message Sender

Any pointers or handles contained within the Data field are assumed to be shared objects.

The WinQueryWindowProcess API call returns the process ID of the receiving process. The sender then uses DosFreeSeg to free up the data in its address.

The sender may free the data calling DosFreeSeg after passing the message to the recipient, but is not required to do so. It may, for example, reuse the same memory for successive transactions.

Syntax of the EXECUTE Command String

DDE provides a means of delivering command strings from a client application to a server application.

An EXECUTE command string is null-terminated. It is recommended that such strings obey the following grammar:

```
<execute string>:= <token><token>...<token>

where:

<token>:= [<opcode string>]

<opcode string>:= <opcode>

or,

<opcode string>:= <opcode><parameter list>

<opcode>:= application-defined single token
           (may not include spaces, commas,
           parentheses, or quotes)

<parameter list>:=(<parameter>,<parameter>...,
                   <parameter>)

<parameter>:= application-defined;
              may not include commas or parens
              except inside a quoted string.
```

Quoted characters, "[", "]", "(", ")", must be doubled if they are to appear as characters within <opcode.string>.

Example EXECUTE strings:

```
[connect][download(query1,results.txt)][disconnect]

[query ("sales per employee for each district")]

[open ("foo.xlm")][run("r1c1")]
```

The System Topic

Applications are encouraged to support at all times a special topic with the name *System*. This topic provides a context for items of information that may be of general interest to any partners in a DDE exchange.

The following items are suggested (this list is not exclusive):

SysItems
: A list of the items supported under the System topic by this application.

Topics
: A list of the topics supported by the application at the current time (this may vary from moment to moment).

ReturnMessage

Supporting detail for the most recently issued WM_DDE_ACK message. Useful when more than 8 bits of application-specific return code are required.

Status

An indication of the current status of the application.

Formats

This item contains a list of clipboard format numbers that the application can render.

Individual elements of such lists should be delimited by tabs (CF_TEXT format).

Part 6. Alphanumeric Text Applications – User Input/Output

Chapter 30. Alphanumeric Video Output

This chapter describes the function calls of the OS/2 Video Subsystem. These calls, which have Vio prefixes, include a variety of functions for alphanumeric output that are written to the logical video buffer as a rectangular array of fixed-size characters.

The OS/2 Video Subsystem provides support for Presentation Manager, text-windowed, and full-screen applications. Some of the calls are specific to an application type and therefore cannot be used by other application types. For example, applications that use the facilities of the Presentation Manager and must direct alphanumeric output to a particular window, need to create a Vio presentation space. The call that performs this function is not appropriate in a text-windowed or full-screen application.

The following sections contain descriptions of the complete set of Vio calls. Section headings indicate the application types supported by the calls.

The OS/2 1.2 Programming Tools do not provide sample programs for all function calls discussed in this chapter. However, when an example of a function call is used in a sample program, the subdirectory path of the sample program is shown. **Note that a sample program may not illustrate the task of the function call as discussed in the chapter text.**

For a quick reference to Vio calls and the application types they support, see Appendix E, "Dos, Kbd, Mou, and Vio Calls by Application Type."

Alphanumeric Support for Presentation Manager Applications

The AVIOSAMP.C sample program provides an example of a simple Presentation Manager text editor that displays alphanumeric video output. This sample program is in the following subdirectory:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP
```

A Presentation Manager application must create a Vio presentation space and associate it with a window before it can write strings to the logical video buffer. Other calls that are specific to an alphanumeric Vio Presentation Manager application include:

VioCreatePS
: Creates a Vio presentation space

WinOpenWindowDC
: Gets a device context handle for the window

VioAssociate
: Associates the device context with the presentation space

VioSetOrg
: Aligns a cell of the presentation space with the origin of the window

WinDefAVioWindowProc
: Maintains alignment of the presentation space with window origin

WinDefWindowProc
: Provides default processing for any window messages not being processed

VioDestroyPS
: Ending the Vio presentation space.

Creating the Vio Presentation Space

Issuing **VioCreatePS** creates the Vio presentation space. A Vio presentation space is a rectangular buffer of alphanumeric characters, which can be displayed in a window. The presentation space can be larger or smaller than the window in which it is displayed.

OS/2 holds a Vio presentation space as a buffer of character and attribute combinations, called cells, in either of the following formats, as specified by your application:

- CGA format
- Extended format.

In CGA format, each character is defined as two bytes. The first byte defines the code point for the character. A code point is the coded representation of a character. The second byte defines attributes of the character and has the following format:

Bit	Attribute
0	Character blue
1	Character green
2	Character red
3	Character intensity
4	Background blue
5	Background green

6	Background red
7	Background intensity.

In extended format, each character is defined as four bytes. The contents of the first two bytes are the same as in CGA format. The third byte defines additional attributes, which are available in this format:

Bit	Attribute
0 – 1	Font identifier
2 – 5	Reserved by OS/2
6	Reverse video
7	Underscore.

An application can reference up to four fonts: the default font and three loadable fonts.

The fourth byte aligns the representation of the character on a word boundary. If required, it can be used by an application.

The maximum size of a presentation space is 64KB. To determine whether your presentation space will exceed this limit, you can use this equation:

```
Number of rows x number of columns x cell length ≤ 64KB
```

where *cell length* equals two bytes in CGA format, and four bytes in extended format.

The number of rows and the number of columns each cannot exceed 255. Each process can have a maximum of 32 presentation spaces; an application can have more than 32 if it consists of more than one process.

Associating the Window with the Presentation Space

After creating a Vio presentation space and a window, get a device context handle for the window by calling **WinOpenWindowDC**. Then associate the device context with the presentation space by calling **VioAssociate**.

To align a particular cell of the Vio presentation space with the origin of the window, call **VioSetOrg**. Whereas the origin of a window is its bottom left corner, the origin of the Vio presentation space is its top left corner.

When a window is sized, **WinDefAVioWindowProc** should be invoked to maintain the alignment between the Vio presentation space and the window origin before the window is accessed. This is a complete default window procedure.

WinDefWindowProc should also be called to provide default processing for any window messages an application chooses not to process.

Setting the Cell Size

Calling **VioSetDeviceCellSize** sets the cell size specified in pels for width and height parameters. The cell sizes supported by the adapters are:

EGA	8 × 8, 8 × 12 (default)
VGA	8 × 8, 8 × 14 (default)
8514/A	7 × 15, 7 × 25, 8 × 8, 8 × 17 (default) 12 × 16, 12 × 20, 12 × 22, 12 × 30.

It is not an error to specify a cell size with VioSetDeviceCellSize that is not supported by the installed adapter. Suppose an adapter supports two sizes, and you specify dimensions for a size that is not supported. If either of the dimensions you request is greater than or equal to either of the respective dimensions of the larger supported size, the larger size is selected. Otherwise, the smaller size is selected. **VioGetDeviceCellSize** can be issued to determine which size is selected.

Any logical fonts created by the application with VioCreateLogFont must be recreated after changing the cell size.

Creating Logical Fonts

The following function calls support creating logical fonts for Presentation Manager applications:

VioQueryFonts
: Queries fonts available for a specified typeface

VioCreateLogFont
: Defines font requirements to enable matches with available fonts

VioQuerySetIds
: Queries logical font definitions created with VioCreateLogFont

VioDeleteSetIds
: Deletes font definitions.

VioQueryFonts returns the number of physical fonts available for a particular typeface, along with their metric descriptions.

VioCreateLogFont creates the logical fonts your application will use in a Vio presentation space. Physical fonts used to satisfy a VioCreateLogFont request can be obtained either from system fonts loaded from the control panel, or from private fonts the process loads with a GpiLoadFonts request.

When you issue VioCreateLogFont, you can specify font dimensions and let the system select whichever of the available physical fonts comes closest to your requirements. Or you can specify a font match value returned by VioQueryFonts and force a particular font's selection. However, only fonts of the correct cell size, as defaulted to or set by VioSetDeviceCellSize, can be used.

Defaults for Text-Windowed Applications

A text-windowed application uses subsets of OS/2 keyboard and mouse function calls for user input (see Chapter 31, "Keyboard and Mouse Input for Text-Windowed and Full-Screen Applications"). A text-windowed application does not create a window, nor does it use messages to communicate. When this type of application runs, OS/2 supplies it with a default window and a default presentation space.

The default window has the following elements in common with the window described in Chapter 5, "The Standard Window" on page 5-1:

- A system icon, from which the user can pull down a system menu
- A title bar, containing the name of the window
- Scroll bars, with which the user can scroll text
- Minimize/maximize/restore icons.

A default window is initially sized. A default logical video buffer of 80 x 25 characters is created, and this is displayed in the window. An application can call VioSetMode to alter the number of rows only.

The default presentation space has a 2-byte character cell format.

Alphanumeric Support for All Application Types

The OS/2 Video Subsystem provides a broad range of function calls for displaying alphanumeric data on the screen. These function calls may be used by Presentation Manager, text-windowed, and full-screen applications.

Depending upon the requirements of your application, you can place alphanumeric data in the buffer, using either of the following methods:

- Directly accessing the logical video buffer with VioGetBuf, manipulating the data, and flushing it to the screen with VioShowBuf. (Presentation Manager applications also use VioShowPS to flush the data.)
- Performing string updates to the logical video buffer with high-function Vio write calls. String updates should be used when only specific parts of the buffer need updating.

Directly Accessing the Logical Video Buffer

The logical video buffer is addressed as a row and column grid of character cells.

The cell format of the logical video buffer (referred to as the Vio presentation space for Presentation Manager applications) depends on the application type:

- Full-screen and text-windowed applications are restricted to the CGA format of a 2-byte character and attribute pair.
- Presentation Manager applications may specify either the CGA format or the 4-byte extended format with VioCreatePS.

Issuing **VioGetBuf** returns a selector and offset for the logical video buffer, and a length. The selector returned is used for accessing the logical video buffer and displaying its contents with VioShowBuf. The length in bytes is calculated by the number of 2-byte or 4-byte character cells the row and column grid can hold. The buffer can be addressed as a word array.

VioShowBuf displays the logical video buffer when the application has the input focus. However, VioShowBuf is ignored unless issued by a process executing in the foreground, that had previously called VioGetBuf.

To determine the amount of memory in the physical video buffer, issue VioGetConfig.

Performing String Updates to the Logical Video Buffer

A set of Vio function calls enable you to write character strings to the logical video buffer (LVB). The calls, which have VioWrt prefixes, support the CGA format of 2-byte character cells and the extended format of 4-byte character cells. The 4-byte format is supported only for Presentation Manager applications. For a description of these cell formats, see "Creating the Vio Presentation Space" on page 30-1.

Vio calls do not go through the OS/2 file system. When you use a VioWrtTTY instead of a DosWrite, you cannot redirect output; however, the VioWrtTTY performs faster.

With Vio character string calls, you do not have to be concerned with accessing the LVB. This is taken care of by the video subsystem.

The following function calls support writing characters to the LVB:

VioWrtTTY
: Writes a string of characters, starting at the current cursor position, and moves the cursor to the position immediately following the string

VioWrtCellStr
: Writes a string of character and attribute combinations

VioWrtCharStr
: Writes a string of characters

VioWrtCharStrAtt
: Writes a string of characters, repeating the specified attribute for each character

VioWrtNChar
: Writes a character a specified number of times

VioWrtNCell
: Writes a character and attribute a specified number of times

VioWrtNAtt
: Writes an attribute a specified number of times.

Writing Teletype Character Strings

When **VioWrtTTY** is issued with a character string, the string is written, Teletype style, starting from the current cursor position. When the write is complete, the cursor is placed in the position immediately following the written string.

The characters for carriage return, line-feed, backspace, tab, and bell are treated as commands, rather than printable characters. The backspace character is a nondestructive backspace, and tab characters are expanded to standard, 8-byte wide fields. VioWrtTTY also recognizes ANSI escape sequence except when the application is AVIO. For AVIO applications, all ANSI escape sequences are ignored.

VioWrtTTY provides the highest level function for character I/O to the logical video buffer. When a VioWrtTTY write extends beyond the end of a line in the LVB, it wraps around and continues on the next line. If the write extends beyond the end of the LVB, the screen is scrolled. Thus, a VioWrtTTY write continues until it is complete. When you use any of the other Vio write calls, the function wraps around to the next line; but if the write is larger than the LVB, the call is terminated.

Sample Programs: Examples of the **VioWrtTTY** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\CRERRCB\CRERRC.C
\TOOLKT12\C\SAMPLES\BSE\DLCBAT\D_INITC.C
\TOOLKT12\C\SAMPLES\BSE\DLCBAT\DLNK1C.C
\TOOLKT12\C\SAMPLES\BSE\HELLOCB\HELLOC.C
\TOOLKT12\C\SAMPLES\BSE\MEMORYCB\MEMORYC.C
\TOOLKT12\MASM\SAMPLES\BSE\CRERRB\CRERR.ASM
\TOOLKT12\MASM\SAMPLES\BSE\DYNBLD\D_INIT.ASM
\TOOLKT12\MASM\SAMPLES\BSE\DYNBLD\DLNK_MG1.ASM
\TOOLKT12\MASM\SAMPLES\BSE\FAMAPI\VIOSAMP.ASM
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB\MEMORY.ASM
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Writing Character Strings and Cell Strings

With the exception of VioWrtTTY, which always writes to the current cursor position, all other Vio character I/O calls can write to any specified row and column position in the logical video buffer.

When a **VioWrtCharStr** is issued, only characters are specified; thus, the characters acquire the existing attributes in the logical video buffer.

Issuing **VioWrtCellStr** allows you to write a string specifying 2-byte or 4-byte character and attribute combinations.

Sample Programs: Examples of the **VioWrtCharStr** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSE.C
\TOOLKT12\C\SAMPLES\BSE\FAMAPIC\VIOSAMPC.C
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB\MEMORY.ASM
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROG1.ASM
```

Replicating Characters, Cells and Attributes

VioWrtCharStrAtt allows you to specify one or more display attributes to be replicated for each character of a specified character string when the string is written.

You can also replicate a character, a cell or an attribute a specified number of times. **VioWrtNChar** writes a repeated character; **VioWrtNCell** writes a repeated cell; and **VioWrtNAtt** writes a repeated display attribute.

Sample Programs: An example of the **VioWrtCharStrAtt** call is found in the following subdirectory:

```
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSE.C
```

Reading Character and Cell Strings from the LVB

Sometimes an application needs to save strings from the logical video buffer. These calls allow you to read characters and save them in a buffer:

VioReadCharStr
: Reads a string of characters

VioReadCellStr
: Reads a string of character and attribute combinations.

Determining the Adapter Type

Issuing **VioGetConfig** returns a display adapter and a display type the Video Subsystem *assumes* is present. The subsystem determines the display type by testing the switch settings on the adapter. However, the display type returned may not be correct, because:

- An adapter may not have a display attached to it
- Adapter switches may not be set correctly for the attached display.

Enabling ANSI Support

VioGetAnsi
: Gets the state of ANSI support

VioSetAnsi
: Sets the state of ANSI support.

VioGetAnsi determines whether ANSI extended keyboard and screen support is turned on or off. The default state for ANSI support is "on," enabling ANSI escape sequences to be entered with a VioWrtTTY function call (see "Writing Teletype Character Strings" on page 30-4). **VioSetAnsi** activates or deactivates ANSI support.

Setting Video Code Pages

Code pages define what characters are displayed on the screen by mapping ASCII code points to character numbers.

Issuing **VioGetCp** returns the current code page, which may have been set with a VioSetCp or inherited by the requesting process.

A return value of 0000 indicates that the active code page is the ROM code page provided by the adapter. The other values that can be returned are either of the two system code pages specified in the user's CONFIG.SYS file or VIO_ERROR_USER_FONT. VIO_ERROR_USER_FONT is a warning that indicates a user font, loaded with VioSetFont, is the active code page.

Note: A value of 0000 can also be returned if no primary system code page has been identified during system initialization, or if there is no DEVINFO statement identifying a video code page in CONFIG.SYS.

In a full-screen or text-windowed application, issuing **VioSetCp** sets the code page to either of the two system code pages defined in CONFIG.SYS or to 0000 for the default ROM code page. A full-screen application can also set the code page to a user font previously loaded with VioSetFont. The displayed characters are reinterpreted immediately in the specified code page.

In a Presentation Manager application, any of the system code pages can be specified with VioSetCp. The characters in the OS/2 font are not reinterpreted in the new code page until other events cause the characters to be repainted.

In a full-screen application, the code page specified with VioSetCp is associated with the specified Vio handle. Since handle zero is the only Vio handle supported for full-screen applications, all threads and processes within a session have to share the same handle and code page.

Requesting a Temporary Foreground Session

VioPopUp and **VioEndPopUp** allow an application to establish a temporary foreground session, called a Vio pop-up. Typically, Vio pop-ups are used for communicating events that require user action to enable processing to continue.

When an application issues VioPopUp, it must wait for a successful return from VioPopUp before proceeding to issue any Vio, keyboard, or mouse calls that are intended for the temporary foreground session of the pop-up. (Vio pop-ups are the only instance where a Presentation Manager application can use keyboard and mouse calls.)

While the pop-up is active, all video calls from the previous foreground session are blocked until the process that issued VioPopUp issues VioEndPopUp.

Only the process within the session that issued VioPopUp is brought to the foreground. If the session was already in the foreground, video calls issued by other processes in the session are blocked until the process ends the pop-up.

A pop-up can wait or return immediately with an error code in the event another pop-up is already in effect.

Vio pop-ups should be used by an application in a manner that is not unnecessarily disruptive to the user.

Sample Programs: Examples of the **VioPopUp** and **VioEndPopUp** calls are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\CRERRCB\CRERRC.C
\TOOLKT12\MASM\SAMPLES\BSE\CRERRB\CRERR.ASM
```

Manipulating the Cursor

The following function calls allow you to alter the position and shape of the cursor in text modes:

VioGetCurPos/VioSetCurPos
: Gets or sets the current position of the cursor

VioGetCurType/VioSetCurType
: Gets or sets the height, width, and attribute of the cursor.

With the exception of VioWrtTTY, Vio character string writes do not change the position of the cursor. Because the function of moving the cursor is separate from writing character strings, your application changes the position of the cursor only when it's necessary, by calling **VioGetCurPos** and **VioSetCurPos**.

To change the shape of the cursor, you can issue **VioSetCurType**. VioSetCurType points to a data structure that defines the following cursor characteristics:

- Starting and ending horizontal scan lines within the character cell
- Width in columns (text mode)
- Attributes.

The maximum number of horizontal scan lines is 32.

The maximum value for the cursor width supported by the OS/2 Video Subsystem in text mode is 1.

A value of −1 for the attribute defines the cursor as hidden. Any other value is normal.

Note: During sizing operations, it is the user's responsibility to maintain the graphical representation of the cursor.

Sample Programs: Examples of the **VioSetCurPos** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\FAMAPIC\VIOSAMPC.C
\TOOLKT12\C\SAMPLES\BSE\MEMORYCB\MEMORYC.C
\TOOLKT12\MASM\SAMPLES\BSE\FAMAPI\VIOSAMP.ASM
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB\MEMORY.ASM
```

An example of the **VioGetCurType** and **VioSetCurType** calls can be found in the following subdirectory:

```
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSE.C
```

Scrolling the Screen Image

The following function calls support scrolling the screen in text modes:

VioScrollUp/VioScrollDn
Scrolls the screen up or down
VioScrollLf/VioScrollRt
Scrolls the screen to the left or to the right.

Vio scrolling calls allow your application to move a specified block of text in the video buffer in any of four directions. In addition, you can specify a fill character to replace the area originally occupied by the moved block.

Sample Programs: Examples of the **VioScrollUp** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\FAMAPIC\VIOSAMPC.C
\TOOLKT12\C\SAMPLES\BSE\MEMORYCB\MEMORYC.C
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSE.C
\TOOLKT12\C\SAMPLES\BSE\VIOCBAT\VIOSAMPC.C
\TOOLKT12\MASM\SAMPLES\BSE\FAMAPI\VIOSAMP.ASM
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB\MEMORY.ASM
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Setting the Display Mode for Text-Windowed and Full-Screen Applications

Display adapters operate in either text or graphics modes. In text modes, the logical video buffer is a row and column grid of addressable character cells. In graphics modes, the logical video buffer becomes all-points-addressable.

The display mode for Presentation Manager applications is set by the Presentation Manager, depending on the type of presentation space you create; when you are using a Vio presentation space, you change the character cell size with VioSetDeviceCellSize. Therefore, the support provided by VioSetMode is not usable in a Presentation Manager application.

VioSetMode can be used by text-windowed and full-screen applications. However, in a text-windowed application, Presentation Manager default window requirements override parameters specified with VioSetMode. In addition, specifying a graphics mode by a text-windowed application causes an error.

When you call VioSetMode, you point to a data structure, which defines the mode. Within this data structure you set a bit mask that contains specifications for the mode. The types listed in the table are the decimal values of these bit masks.

Following are the meanings of the bit settings:

Bit	Setting
0	0 = monochrome compatible; 1 = other
1	0 = text mode; 1 = graphics mode
2	0 = enable color burst; 1 = disable color burst
3	0 = VGA compatible modes, 0 – 13H; 1 = native mode
4 – 7	Reserved.

For a text-windowed application, monochrome compatibility and color burst bit settings are ignored, and an attempt to set a graphics mode is rejected.

In full-screen applications, the color burst bit is functional for the color graphics adapter (CGA) and the video graphics array (VGA). The color burst bit for the CGA disables color for composite video only; a gray scale palette is loaded to simulate this feature on the VGA. For all other display adapters, the setting of this bit is recorded and returned on any subsequent **VioGetMode** call but is otherwise ignored.

For a text-windowed application, color parameters and horizontal and vertical resolution parameters are ignored. The text columns parameter must be set to 80. The only parameter of VioSetMode that offers a choice to a text-windowed application is the number of rows, which can be set to any value from 1 to 255. Full-screen applications can be set to one of the following, depending on the adapter installed:

Adapter	Rows
CGA	25
EGA	25 or 43
VGA	25, 30, 43, 50, or 60
8514/A	25, 30, 43, 50, or 60.

To set the mode, initialize the data structure and issue VioSetMode. VioSetMode initializes the current settings for cursor position and type; however, it does not clear the screen. To clear the screen, you must issue one of the VioScrollXX calls. See "Scrolling the Screen Image." To reset the cursor position and type, see "Manipulating the Cursor" on page 30-6.

Table 30-1 on page 30-8 lists the OS/2 display modes that are compatible with BIOS modes, and equivalent BIOS modes if any. When VioSetMode is issued by a text-windowed application, it does not cause a change of character cell size, as it does for a full-screen application. Instead, the user can toggle between two character cell sizes (in the case where two cell sizes are supported) from the system menu.

Table 30-1 (Page 1 of 2). Display Mode Attributes Supported by Adapters

BIOS Mode	Type	Color	Cols	Rows	HRes	VRes	Valid Adapter/Display Combinations [Emulated]
0	5	4	40	25	320	200	[CGA/CD], CGA/Comp, [EGA/CD], [EGA/ECD], VGA/Mono, VGA/Color, VGA/Plasma
0*	5	4	40	25	320	350	[EGA/ECD], VGA/Mono, VGA/Color, VGA/Plasma
0+	5	4	40	25	360	400	VGA/Mono, VGA/Color
0#	5	4	40	25	320	400	VGA/Mono, VGA/Color, VGA/Plasma
1	1	4	40	25	320	200	CGA/CD, CGA/Comp, EGA/CD, EGA/ECD, [VGA/Mono], VGA/Color, [VGA/Plasma]
1*	1	4	40	25	320	350	EGA/ECD, [VGA/Mono], VGA/Color, [VGA/Plasma]
1+	1	4	40	25	360	400	[VGA/Mono], VGA/Color
1#	1	4	40	25	320	400	[VGA/Mono], VGA/Color, [VGA/Plasma]
2	5	4	80	25	640	200	[CGA/CD], CGA/Comp, [EGA/CD], [EGA/ECD], VGA/Mono, VGA/Color, VGA/Plasma
2*	5	4	80	25	640	350	[EGA/ECD], VGA/Mono, VGA/Color, VGA/Plasma
2+	5	4	80	25	720	400	VGA/Mono, VGA/Color
2#	5	4	80	25	640	400	VGA/Mono, VGA/Color, VGA/Plasma
3	1	4	80	25	640	200	CGA/CD, CGA/Comp, EGA/CD, EGA/ECD, [VGA/Mono], VGA/Color, [VGA/Plasma]
3*	1	4	80	25	640	350	EGA/ECD, [VGA/Mono], VGA/Color, [VGA/Plasma]
3+	1	4	80	25	720	400	[VGA/Mono], VGA/Color
3#	1	4	80	25	640	400	[VGA/Mono], VGA/Color, [VGA/Plasma]
7	0	0	80	25	720	350	MPA/MD, EGA/MD, VGA/Mono, VGA/Color
7+	0	0	80	25	720	400	VGA/Mono, VGA/Color
7#	0	0	80	25	640	400	VGA/Mono, VGA/Color, VGA/Plasma
n/a	0	0	80	25	640	350	VGA/Mono, VGA/Color, VGA/Plasma
n/a	1	4	80	30	720	480	[VGA/Mono], VGA/Color
n/a	1	4	80	30	640	480	[VGA/Mono], VGA/Color, [VGA/Plasma]
4	3	2	[40]	[25]	320	200	CGA/CD, CGA/Comp, EGA/CD, EGA/ECD, [VGA/Mono], VGA/Color, [VGA/Plasma]
5	7	2	[40]	[25]	320	200	[CGA/CD], CGA/Comp, [EGA/CD], [EGA/ECD], VGA/Mono, VGA/Color, VGA/Plasma
6	3	1	[80]	[25]	640	200	CGA/CD, CGA/Comp, EGA/CD, EGA/ECD, VGA/Mono, VGA/Color, VGA/Plasma
D	3	4	[40]	[25]	320	200	EGA/CD, EGA/ECD, [VGA/Mono], VGA/Color, [VGA/Plasma]
E	3	4	[80]	[25]	640	200	EGA/CD, EGA/ECD, [VGA/Mono], VGA/Color, [VGA/Plasma]
F	2	0	[80]	[25]	640	350	EGA/MD, VGA/Mono, VGA/Color, VGA/Plasma
10	3	4	[80]	[25]	640	350	EGA/ECD, [VGA/Mono], VGA/Color, [VGA/Plasma]
11	3	1	[80]	[30]	640	480	VGA/Mono, VGA/Color, VGA/Plasma
12	3	4	[80]	[30]	640	480	[VGA/Mono], VGA/Color, [VGA/Plasma]
13	3	8	[40]	[25]	320	200	[VGA/Mono], VGA/Color, [VGA/Plasma]
n/a	11	8	[80]	[30]	640	480	[8514A/Mono], 8514A/Color
n/a	11	4	[80]	[30]	640	480	[8514A/Mono], 8514A/Color

Table 30-1 (Page 2 of 2). Display Mode Attributes Supported by Adapters

BIOS Mode	Type	Color	Cols	Rows	HRes	VRes	Valid Adapter/Display Combinations [Emulated]
n/a	11	8	[85]	[38]	1024	768	[8514A/HMono], 8514A/HColor
n/a	11	4	[85]	[38]	1024	768	[8514A/HMono], 8514A/HColor

Display Adapters:

MPA	Monochrome/Printer Adapter
CGA	Color Graphics Adapter
EGA	Enhanced Graphics Adapter
VGA	Video Graphics Array, PS/2 Display Adapter
8514A	8514/A Display Adapter

Displays:

MD	5151 Monochrome Display
CD	5153 Color Display
ECD	5154 Enhanced Color Display
Mono	8503 PS/2 Monochrome Display, 8507/8604 Display
HMono	8507/8604 Display
Color	8512/13 PS/2 Color Display, 8514 Display
HColor	8514 Display
Plasma	Plasma Display Panel
Comp	Composite Video Monitor

Notes:

1. Types 0, 1, and 5 are text modes; types 2, 3, 7, and 11 are graphics modes.
2. For BIOS modes 0, 2, 5, the color burst is disabled on the CGA and VGA.
3. The Personal System/2 Display Adapter 8514/A has advanced function modes, which are supported through the 8514/A display adapter interface, not the VIO Subsystem. Refer to the *Personal System/2 Display Adapter 8514/A Technical Reference* for details of this support.

Video Output Support Specific to Full-Screen Applications

The function calls described in the remaining sections of this chapter are intended for use only by full-screen applications and cannot be used by Presentation Manager or text-windowed applications.

Setting the State of the Adapter

The **VioSetState** call can issue any one of the following request types:

- Set the palette registers for EGA, VGA, and PS/2 adapters
- Set the border color on CGA, VGA and PS/2 adapters
- Enable blinking foreground colors or high intensity background colors for CGA, EGA, VGA, and PS/2 adapters
- Set the color registers for VGA or PS/2 adapters
- Set the display mask for the PS/2 display adapter 8514/A
- Set the scan line for underlining for EGA, VGA, and PS/2 adapters
- Set the VioSetMode display configuration.

Note that request type = 6, VioSetMode display configuration, and request type = 5, set scan line for underlining, are not supported by family API.

A data structure is pointed to, whose contents are dependent on the request type. **VioGetState** returns the current values for these functions in the data structure.

Loading Fonts

VioGetFont points to a data structure that contains a request to return the far address of the current font or a font stored in the adapter's ROM.

If a ROM font is requested, the font size specified must be supported by the adapter installed:

Adapter	Font Sizes Supported
VGA, PS/2	8x8, 8x14, 9x14, 8x16, 9x16
EGA	8x8, 8x14, 9x14
CGA	8x8.

A ROM pointer is returned when the font table for the full 256-character set is contained in ROM; otherwise, a RAM pointer is returned. For font sizes 8x8, 9x14, and 9x16, a pointer to the full character set in ROM is returned. The 9x14 and 9x16 fonts are derived from variations of the 8x14 and 8x16

fonts. The fonts for those characters that are different are replaced.

Fonts are derived from fonts contained in the system and in the BIOS data area of the EGA, VGA, or PS/2 adapter. However, if a non-zero codepage is active, the font size requested from the active codepage is returned.

When issuing **VioSetFont** to download a user font as the current font, the character cell defined for the adapter by its mode setting must agree with the font size specified in VioSetFont. If the font size differs, the font will be saved for use with a later VioSetMode. Refer to Table 30-1 on page 30-8.

When VioSetFont is called, the active code page is disabled. Thus, issuing a subsequent VioGetCp returns an error message.

Video Font File Organization

A system video font file contains video font tables for all the code pages supported by OS/2. When needed, a font table is extracted from this file by the video subsystem.

The format of the video font file header is shown in Figure 30-1.

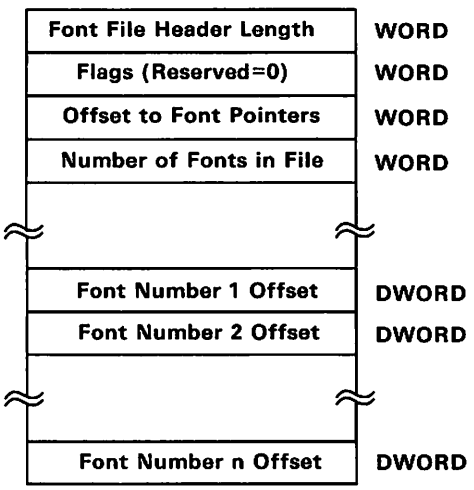


Figure 30-1. Video Font File Header Format

Figure 30-2 illustrates the format of a font table. The Font Flags field indicates whether a font table contains a partial font and if code points for characters are included. The format of the Font Flags field is:

Bit 0 = 1 Partial font
Bit 1 = 1 Code points included
Bit 2 – 15 Reserved; zeros.

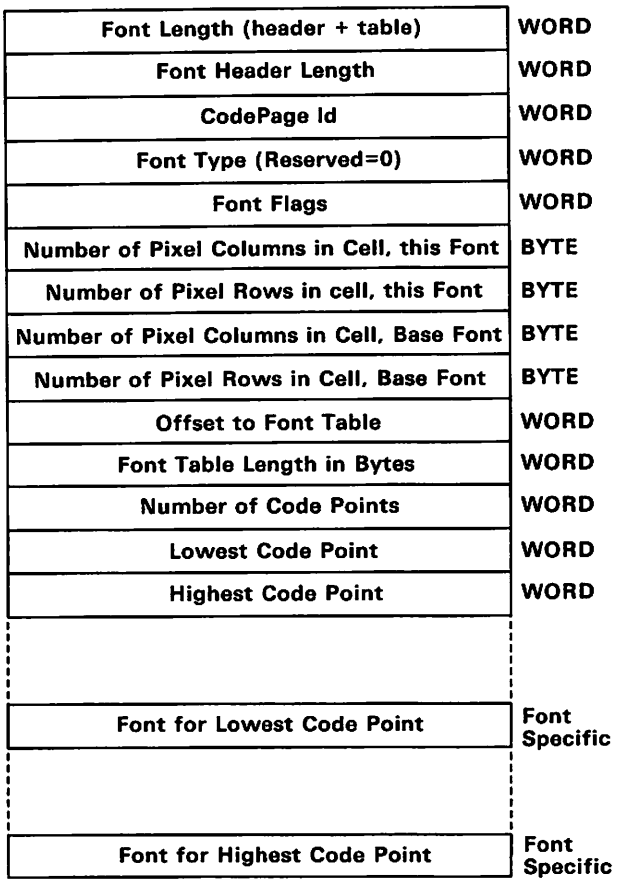


Figure 30-2. Video Font File Table Format

Using Your Own Graphics Interfaces

Typically, there is no need for applications to access the physical video buffer. However, this is appropriate when you are writing graphics routines for a full-screen application that directly access the registers of the display adapter.

The following function calls support accessing the physical video buffer:

VioGetPhysBuf
: Obtains one or more selectors to access the physical video buffer

VioScrLock
: Requests ownership of the physical video buffer

VioScrUnLock
: Relinquishes ownership of the physical video buffer.

Only OS/2 full-screen applications may use these calls.

Obtaining Addressability to the Physical Buffer

VioGetPhysBuf returns one or more 16-bit selectors that are used to address the physical video buffer. The selectors can be used only when an application is running in the foreground. To determine whether the application is in the foreground, you must issue VioScrLock.

The first selector addresses either the first 64KB of the buffer, or the length of the buffer, whichever is smaller. If the length of the buffer exceeds 64KB, then a second selector is returned to address the next 64KB.

You must initialize the first doubleword of the data structure where the selectors are returned to the 32-bit physical address of the display adapter's buffer. The next doubleword is the length of the buffer you wish to access. Be sure that you provide enough space in the data structure to accommodate all the selectors.

Selectors for the physical buffer cannot be used during a Vio pop-up.

Note: To determine the amount of memory in the adapter, you can issue VioGetConfig.

Requesting Ownership of the Physical Buffer

VioScrLock can be issued with a request to wait until the physical buffer is available or to return immediately, so any processing can continue in the background.

Once a VioScrLock is in effect, screen switching is disabled for a system-defined amount of time, starting from the time of a screen switch request. When this time limit is up, the screen switch occurs, and the process that issued the lock becomes frozen in the background.

VioScrUnlock relinquishes ownership of the physical buffer. This call must be issued by a thread within the same process that issued VioScrLock.

Saving and Restoring the Screen Context and Video State

Usually, when a full-screen application running in a text mode is switched from the foreground to the background or from the background to the foreground, OS/2 handles the saving and restoring of the screen context. Applications do not normally access the adapter's registers directly in a text mode. However, whenever a full-screen application is writing directly to the registers, whether in a text mode or in a graphics mode, it assumes the responsibility for saving and restoring the screen context.

The following function calls support save and restore operations for full-screen applications:

VioSavRedrawWait
: Waits to save or redraw the physical video buffer

VioSavRedrawUndo
: Cancels the waiting of another thread that issued VioSavRedrawWait

VioModeWait
: Waits to restore the video mode, state, and display adapter registers

VioModeUndo
: Cancels the waiting of another thread that issued VioModeWait.

VioSavRedrawWait is issued by a thread that is dedicated to this call. The thread's function is to wait for notification by the Vio subsystem that a screen switch request has occurred. It can then save or restore the screen image, depending on whether the application is in the foreground or background. The thread allocates a buffer for this purpose.

VioModeWait is issued by another dedicated thread in the same process, whose function is to wait for notification to restore the video mode. An application must issue VioModeWait only if it writes directly to the registers on the display adapter. Otherwise, the application can allow OS/2 to perform the required restore by not issuing VioModeWait.

The following events trigger save or restore operations:

1. A user requests a switch to another application.
2. An application issues DosStartSession, specifying the foreground.
3. An application issues DosSelectSession.
4. A Vio pop-up is issued by an application or the system for a hard error.
5. A Vio pop-up ends.
6. The foreground session ends.

To be notified for events 1 and 2, the application issues VioSavRedrawWait. Upon the return of VioSavRedrawWait, the application is notified whether it is to perform a save or a restore.

When an application is notified to perform a save or a restore, it must save or restore the contents of the physical display buffer, the video mode, state, and any other modified display adapter registers and immediately reissue VioSavRedrawWait.

At the instant the VioSavRedrawWait thread is notified to perform a save or a restore, the session is in transition to or from the background. Although the official status of the session may be the background, any selector to the physical video buffer returned by VioSavRedrawWait is still valid. The physical buffer must be accessed without issuing VioScrLock. If VioScrLock were to be issued with "wait if unsuccessful," it would wait because the session is officially still in the background.

Note: Be aware that a user can start an application in the background with the START command. If an application is started in the background and VioSavRedrawWait is issued, the thread can be notified to perform a restore before it performs a save.

An application is not notified for event 4, because OS/2 automatically saves and restores the contents of the physical display buffer in the area reserved for pop-ups.

To be notified for event 5, the application issues VioModeWait. A graphics mode application must have a VioModeWait thread if it is writing directly to the display registers. When a VioPopUp is issued, OS/2 changes the mode to text. The mode must be restored by the application when the pop-up ends. The return of VioModeWait is the notification to restore the mode, state, and display registers. Upon doing so, the application must immediately reissue VioModeWait.

VioSavRedrawWait and VioModeWait can be issued only once during a foreground or background session. The first process of a multiple process application to issue these calls becomes the owner. Issuing threads can reserve ownership, or their ownership can be canceled by issuing **VioSavRedrawUndo** and **VioModeUndo**. These calls must be issued by another thread of the process that owns the waiting threads.

Registering Another Video Subsystem

Like any OS/2 I/O subsystem, the OS/2 video subsystem consists of subroutines that can be individually replaced.

The following function calls register and deregister an alternate video subsystem:

VioRegister
: Registers a video subsystem for the current session

VioDeRegister
: Cancels the registration of a video subsystem

VioGlobalReg
: Globally registers a video subsystem.

Only OS/2 full-screen applications may use these calls.

Individual replacement of video function calls is accomplished by setting bits in a function call bit map specified with **VioRegister**. Once the alternate video subsystem is registered, and a Vio call is issued, OS/2 checks the bit map for the call.

If a bit is set for the call, indicating a replacement for the function, the alternate dynamic link module is called. Control is passed to the entry point name of the routine specified with VioRegister.

If a replacement routine for VioPopUp has been registered, the routine is called only when the session is in the foreground. If VioPopUp is issued when the session is in the background, the OS/2 default routine is called.

All video functions within a session are serialized on a thread basis. For example, when an alternate video subsystem receives control, it can safely assume that it will not be called again from the same session until the current routine is complete.

VioRegister remains in effect within the session until a **VioDeRegister** is issued or the session ends. VioDeRegister must be called by the same process that issued VioRegister.

VioGlobalReg allows a subsystem to receive notification at the completion of VIO calls issued by all full-screen sessions. Notification of VIO calls issued within the hard error handler and DOS (real mode) sessions is not provided. VioGlobalReg is issued during system initialization. After system initialization, VioGlobalReg will return ERROR_VIO_REGISTER. A globally registered subsystem is active for the life of the system.

Note: Control is given to multiple global subsystems in the order in which they are registered.

Sharing the Vio Handle with Other Processes and Threads

Unlike Presentation Manager applications, full-screen applications use only the Vio zero handle. Thus, if an application has multiple threads and processes, they all must share the same handle. Vio calls issued by multiple threads and processes within a session are serialized with a semaphore by the video subsystem. Potential deadlocks can occur when Vio calls compete for the semaphore. Following are two examples.

Example 1: Suppose a thread enters a critical section. The thread issues a Vio call at the same time that a Vio call by another thread allows the Vio semaphore to be claimed. This causes the thread in the critical section to fail.

Example 2: Suppose a thread issues a VioPopUp and attempts to claim a semaphore. The semaphore is already owned by a thread of another process in the session, and the thread is blocked on a Vio call because of the Vio pop-up. This causes the thread of the first process to fail.

Refer to Chapter 35, "Communications Between Threads and Processes" for a description of semaphores, and to Chapter 34, "Program Execution and Control" for a description of the DosEnterCritSec function call.

Chapter 31. Keyboard and Mouse Input for Text-Windowed and Full-Screen Applications

This chapter describes the two sets of function calls used for accessing the keyboard and mouse devices, which can be used by text-windowed and full-screen applications. **These calls cannot be used in a Presentation Manager application.**

If you are writing a full-screen application, which does not use the facilities of the Presentation Manager, there are no restrictions on the keyboard and mouse calls your application can use.

If your application is intended as a text-windowed application, which is supplied a default window by the Presentation Manager when it runs, you must use subsets of the keyboard and mouse calls to process user input.

OS/2 establishes a default keyboard for each text-windowed application. All the keyboard calls can be used, except KbdRegister and KbdDeRegister. However, KbdSetStatus and KbdGetStatus function in a limited way.

Mouse calls receive input only when the default window has the input focus. When the user selects the window of a text-windowed application, OS/2 brings it to the front of the viewing order. All the mouse calls can be used, except MouRegister and MouDeRegister. However, the following calls work in a limited way:

MouOpen	**MouClose**
MouDrawPtr	**MouRemovePtr**
MouGetDevStatus	**MouSetDevStatus**
MouGetPtrPos	**MouSetPtrPos**
MouGetPtrShape	**MouSetPtrShape**
MouGetScaleFact	**MouSetScaleFact.**

The OS/2 1.2 Programming Tools do not provide sample programs for all function calls discussed in this chapter. However, when an example of a function call is used in a sample program, the subdirectory path of the sample program is shown. **Note that a sample program may not illustrate the task of the function call as discussed in the chapter text.**

Keyboard Input

The OS/2 Keyboard Subsystem consists of function calls with Kbd prefixes and a keyboard device driver that is installed by OS/2.

Just as video function calls have a file system counterpart in DosWrite, so do keyboard function calls in DosRead. Your decision to use DosRead or keyboard function calls depends on the immediate requirements of your application. For example, when your application needs redirection capabilities, use DosRead. And when your application needs to examine scan codes of ASCII character input, use KbdCharIn.

Keyboard function calls that specify a handle for the logical keyboard complete their processing when the handle has the input focus, or when the handle is equal to 0 and no other handle has the input focus. The input focus is obtained with **KbdGetFocus.**

Reading Keyboard Input

Keyboard function calls allow you to read character input as strings or individual character records. The following function calls support reading input from the logical keyboard buffer:

KbdStringIn
: Reads a string of character codes from a logical keyboard buffer

KbdCharIn
: Reads a character data record from a logical keyboard buffer

KbdPeek
: Examines a character data record without removing it from the buffer

KbdXlate
: Translates a character data record's scan code and shift key state into an ASCII character code

KbdFlushBuffer
: Clears a logical keyboard buffer.

Reading a String of ASCII Character Codes

KbdStringIn reads a string of character codes from the logical keyboard buffer. The operation of KbdStringIn depends on the current keyboard input mode that has been set for the keyboard subsystem. Keyboard input modes can be queried and set with KbdGetStatus and KbdSetStatus (see "Setting the Keyboard State" on page 31-3). There are three input modes:

- ASCII
- Binary
- Echo.

In ASCII mode, which is the default, the carriage return character is recognized. Specify the "IOWait" parameter to have KbdStringIn return when it reads a carriage return character. Because ASCII mode supports synchronous user input, the "no wait" parameter is not applicable.

The received input length is used in ASCII mode by KbdStringIn to support line editing function keys pressed by the user. On the next KbdStringIn, the received input length indicates the length of the buffer that can be recalled using these keys. A value of 0 inhibits the line editing function for the current KbdStringIn request.

In ASCII mode, DBCS characters and extended ASCII character codes are returned in complete form as 2-byte character strings. For an extended ASCII character code, the first byte is 00H or E0H, and the second byte is the extended ASCII code.

In binary mode, the carriage return character has no special meaning. When "IOWait" is specified, KbdStringIn waits until the specified buffer length is full. If the "no wait" parameter is specified, KbdStringIn returns immediately when no characters are available.

In echo mode, characters are echoed from the buffer to the display as the user enters them. Echo mode can be set in conjunction with ASCII mode; however, binary and echo modes cannot be used together.

Sample Programs: Examples of the **KbdStringIn** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\VIOBAT\VIOSAMPC.C
\TOOLKT12\MASM\SAMPLES\BSE\VIOSAMPB\VIOSAMP.ASM
```

Reading Individual Character Data Records

KbdCharIn reads the data record for a key that has been pressed and returns information to a character data structure that has been pointed to. This information includes an ASCII character code, a scan code, character status, and keyboard shift state.

The ASCII code field contains the ASCII character code, which the the keyboard device driver obtains by looking up the scan code generated by the keyboard in a keyboard layout translation table. A scan code is the arbitrary number assigned to a particular key. If a key has been pressed that has an extended ASCII character code and no ASCII equivalent, the ASCII code field is set to 00H or E0H.

In most cases, the scan code field contains the scan code generated by the keyboard. However, in the case where a key has an extended ASCII character code, the scan code field contains the extended ASCII character code.

The status field defines whether the character is a single-byte character or a DBCS character. If the character is a DBCS character, this field identifies which byte of the character has been received, the first (interim) byte or the second.

When KbdCharIn is issued with the "IOWait" parameter set to 0, the calling thread waits until a character becomes available. If "IOWait" is set to 1, KbdCharIn returns immediately when no characters are available.

Setting "IOWait" to 1 to poll the keyboard should be avoided. A thread in the foreground session that polls the keyboard can prevent all regular priority class threads from executing. If polling must be used and a minimal amount of other processing is being performed, the thread should periodically yield processing time to the other threads by issuing DosSleep for an interval of at least 5 milliseconds.

Sample Programs: Examples of the **KbdCharIn** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\CRERRCB\CRERRC.C
\TOOLKT12\MASM\SAMPLES\BSE\CRERRB\CRERR.ASM
```

Peeking at a Character Data Record

KbdPeek allows you to examine a character data record without removing it from the buffer. KbdPeek uses the same data structure as KbdCharIn.

Note: Do not use KbdPeek to poll the keyboard.

Translating a Character Data Record

KbdXlate translates a scan code and shift key state into an ASCII character code, using the code page set for the keyboard. KbdXlate can be called repeatedly to complete the translation of a character that has been entered with complex key combinations, such as an accented character.

Note: Do not use KbdXlate as a replacement for the OS/2 system keystroke translation function.

Clearing the Keyboard Input Buffer

KbdFlushBuffer clears the keyboard input buffer of all queued key strokes. Depending on the requirements of your application, you may want to issue KbdFlushBuffer before you read input from the keyboard.

Sample Programs: Examples of the **KbdFlushBuffer** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\CRERRCB\CRERRC.C
\TOOLKT12\MASM\SAMPLES\BSE\CRERRB\CRERR.ASM
```

Accessing a Secondary Logical Keyboard

Each application running in a session has a default logical keyboard. An application may have more than one thread that performs console I/O. To separate keyboard input of individual threads executing within a session, a secondary logical keyboard is created.

The following function calls support using a secondary logical keyboard for input:

KbdOpen/KbdClose
: Opens and closes a handle to a secondary logical keyboard

KbdGetFocus/KbdFreeFocus
: Gets and releases the input focus by the secondary keyboard.

Issuing **KbdOpen** initializes a secondary logical keyboard to the system code page currently in use and returns a keyboard handle that identifies the logical keyboard.

To bind the secondary logical keyboard with the physical keyboard, call **KbdGetFocus**. If the physical buffer is not in use by another logical or default keyboard, the bind is made. Otherwise, the thread that issued KbdGetFocus may wait until the input focus is freed. Waiting threads do not execute in any definable order.

After I/O is performed with the keyboard handle, issue **KbdFreeFocus** to release the physical buffer for other threads to use. If there are no threads waiting, the physical keyboard is returned to the default keyboard (handle 0).

Managing the Keyboard Device

The following function calls support managing the keyboard device:

KbdGetStatus/KbdSetStatus
: Gets and sets the keyboard state

KbdGetHWid
: Queries the keyboard hardware identification value

KbdGetCP/KbdSetCP
: Gets and sets the ID of the system code page used to translate scan codes into ASCII codes

KbdSetCustXt
: Installs a customized keyboard translation table.

Setting the Keyboard State

The initial state of the keyboard is established when the application is loaded. **KbdSetStatus** is used to modify the keyboard default state.

As mentioned in the discussion of KbdStringIn, you can set keyboard input modes with KbdSetStatus. These modes are described in "Reading a String of ASCII Character Codes" on page 31-2.

Setting a bit for a shift report enables your application to retrieve separate data packets for shift state changes with KbdCharIn. A shift report is not valid in ASCII mode.

You can define the turnaround character as any single or double-byte character. The default turnaround character is the carriage return (0DH).

Bits can be set to alter the shift state of any of the various keyboard shift keys.

The interim character flag enables or disables the processing of DBCS characters.

The file system function call DosRead uses the keyboard subsystem to read input from the keyboard. Thus, the keyboard state set with KbdSetStatus is in effect for any DosRead calls that read input from the keyboard.

Text-Windowed Considerations: If bits are set for shift report, shift state, or interim character flag, no operations are performed, and a return code of 0 is returned.

Querying Keyboard Hardware ID

This function is of particular usefulness for applications providing custom translate tables and mapping keyboard layouts. **KbdGetHWid** will return the attached keyboard's hardware generated identification value.

In the past, all keyboards were supported by the hardware family information via keyboard IOCtl 77H. With the addition of the 122-key keyboard, recognition is not obtainable by hardware family information alone. The 122-key keyboard has a number of differences from other keyboards. Therefore, applications performing keystroke-specific functions may need to determine specifically which keyboard is attached.

Setting the Code Page for the Keyboard

Code pages define the characters that are entered from the keyboard by mapping ASCII code points to character numbers. Code page support by the keyboard subsystem is equivalent to the code page support provided by the video subsystem. **KbdSetCP** sets the code page to either of the two system code pages defined in CONFIG.SYS or to 0000 for the keyboard's internal code page.

Issuing **KbdGetCP** returns the current code page, which may have been set with a KbdSetCP or inherited by the requesting process. If a value of 0000 is returned, the active code page is the ROM code page provided by the keyboard. The other values that can be returned are either of the two system code pages specified in the user's CONFIG.SYS file.

Note: A value of 0000 can also be returned if no primary system code page is identified during system initialization, or if there is no DEVINFO statement identifying a keyboard code page in CONFIG.SYS.

Supporting a Customized Keyboard Layout

The keyboard device driver associates a scan code with a key by looking up the scan code in a keyboard-specific translation table. OS/2 provides translation tables for standard keyboard layouts. To provide support for a customized keyboard layout within the current session, you can load a keyboard translation table using **KbdSetCustXt**.

Registering Another Keyboard Subsystem

Like any of the OS/2 I/O subsystems, the keyboard subsystem consists of subroutines that can be individually replaced.

The following function calls register and deregister an alternate keyboard subsystem:

KbdRegister
: Registers a keyboard subsystem for the current session

KbdDeRegister
: Cancels the registration of a keyboard subsystem

KbdSynch
: Synchronizes the subsystem's access to the physical keyboard.

These calls cannot be used by a text-windowed application. (Of course, **none** of the keyboard calls can be used by a Presentation Manager application.)

Replacing Keyboard Function Calls

Individual replacement of keyboard function calls is accomplished by setting bits in a function call bit map specified with **KbdRegister**. Once the alternate keyboard subsystem is registered, and an OS/2 keyboard call is issued, the system checks the bit map for the call.

If a bit is set for the call, indicating a replacement for the function, the alternate dynamic link module is called. Control is passed to the entry point name of the routine specified with KbdRegister.

KbdRegister remains in effect within the session until a **KbdDeRegister** is issued or the session ends. KbdDeRegister must be called by the same process that issued KbdRegister.

Synchronizing Keyboard Access

KbdSynch is intended for use by an alternate keyboard subsystem to synchronize access between the subsystem and the keyboard device driver. An exclusive semaphore is used by KbdSynch, which blocks other threads in the session until the semaphore is cleared. The semaphore is cleared when the subsystem returns to the keyboard router.

To ensure synchronization, KbdSynch should be issued when the alternate keyboard subsystem intends to issue DosDevIOCtl or access dynamically modifiable per-session shared data. KbdSynch will not protect globally shared data from threads in other sessions.

Mouse Input

The OS/2 Mouse Subsystem consists of function calls with Mou prefixes and two device drivers, the pointer draw device driver and the mouse device driver, which must be specified by the user with DEVICE commands in the CONFIG.SYS file.

Mouse input consists of data records that report the occurrence of events such as mouse movement and the pressing or releasing of a button. Movement is measured in either column and row coordinates or mickeys. (A mickey is a unit of measurement for physical mouse motion, whose value depends on the mouse device driver currently loaded.)

Each session has a physical mouse event queue. When the queue becomes full, the oldest records in the queue are overwritten by new data coming in.

Opening the Mouse Event Queue

An application issues **MouOpen** to initialize the mouse event queue and obtain a handle to access it. The initial state of the mouse device is:

- All events are reported.
- Row and column scale factors are set to 16×8.
- The event queue is empty.
- All user-settable device status bits are reset.
- The pointer is set to the center of the screen.
- The pointer shape is set to the pointer device driver's default.
- The pointer image restricted area is defined as the screen size.

Only the first of a series of multiple MouOpen calls affects or resets the mouse environment.

Issuing **MouClose** closes the mouse device for the current session and removes the mouse device driver handle from the list of valid open mouse device handles.

Text-Windowed Considerations: The handle returned by MouOpen is unique for the application, not the screen group as in the base operating system. This handle is used in future mouse calls.

MouClose closes the locator device for the current application. After this call is issued, the handle is invalid.

Sample Programs: Examples of the **MouOpen** and **MouClose** calls are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSEC.C
\TOOLKT12\MASM\SAMPLES\BSE\MOUSAMPB\MOUSAMP.ASM
```

Setting the State of the Event Queue

The following function calls support setting the state of the mouse:

MouGetNumButtons
: Determines the number of buttons supported

MouGetEventMask/MouSetEventMask
: Gets or sets the types of events reported by data records

MouSetDevStatus
: Sets mouse data to be returned in mickeys instead of coordinates

MouGetNumMickeys
: Determines the number of mouse motion units per centimeter

MouGetScaleFact/MouSetScaleFact
: Gets or sets the mickey-to-pel ratio for mouse motion

MouGetDevStatus
: Gets the state of the pointer and the event queue.

Determining the Type of Mouse Events to Be Recorded

Issuing **MouSetEventMask** enables an application to select the mouse events for which data records are placed in the event queue. Bits are set for these events, which indicate mouse transition states to be recorded. For example, when bit 3 is set, the transitions of pressing or releasing button 2 while moving the mouse are recorded.

Sample Programs: Examples of the **MouSetEventMask** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSEC.C
\TOOLKT12\MASM\SAMPLES\BSE\MOUSAMPB\MOUSAMP.ASM
```

Before you set the event mask, you can determine the number of mouse buttons supported by the installed mouse device driver by calling **MouGetNumButtons**. **MouGetEventMask** queries the value of the event mask.

Returning Mouse Data in Mickeys

Normally, mouse data is returned to the application with the absolute display mode coordinates of the pointer image position on the display screen. By setting a flag with MouSetDevStatus, you can have mouse data returned in mickeys.

To determine the number of mickeys per centimeter for the installed mouse device driver, issue **MouGetNumMickeys**.

MouSetScaleFact is called to set row and column scaling factors for mouse motion. Scaling factors are ratios of the number of mickeys required to move the pointer 8 pels on the screen. These ratios define the sensitivity of the mouse and determine the rate at which the pointer moves on the screen. Ratio default values are:

Row 8 mickeys to 8 pels
Column 16 mickeys to 8 pels.

The number of pels moved does not have to correspond one-to-one with the number of mickeys moved.

Text-Windowed Considerations: **MouGetScaleFact** always returns the current scaling factor set by the Presentation Manager, which is a row and column scaling factor of 1 character.

When MouSetScaleFact is called, no operation is performed and a return code of 0 is returned.

Querying the State of the Pointer and Event Queue

Calling **MouGetDevStatus** returns the settings for status flags that indicate the following states:

- Mickeys or absolute screen coordinates are used.
- Pointer draw device driver is called.
- An unsupported display mode is being used.
- A read is in progress.
- A buffer flush is in progress.
- Event queue is busy with I/O.

The first two of these states can be reset.

By setting a status flag with **MouSetDevStatus**, an application can request that the mouse device driver not call the pointer draw device driver. The application assumes the responsibility for drawing the pointer image on the screen.

An application can also request that data be returned in mickeys instead of absolute screen coordinates.

Text-Windowed Considerations:
MouGetDevStatus always returns a word of zeros, and no operation is performed for MouSetDevStatus.

Reading Data from the Event Queue

The following function calls support reading data records from the mouse event queue:

MouReadEventQue
Reads a data record from the event queue
MouGetNumQueEl
Determines the number of records in the queue
MouFlushQue
Clears the queue.

Reading a Data Record from the Event Queue

MouReadEventQue reads a data record from the FIFO event queue and places it in a 10-byte data structure provided by the application. If the queue is empty, the application can wait until data is placed in the queue, or it can return with a null record.

The data records placed in the queue depend on the types of events that have been set with MouSetEventMask. When the queue becomes full, the oldest records are overwritten.

Sample Programs: Examples of the **MouReadEventQue** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSEC.C
\TOOLKT12\MASM\SAMPLES\BSE\MOUSAMPB\MOUSAMP.ASM
```

Determining the Number of Records in the Event Queue

MouGetNumQueEl returns the number of elements in the mouse device driver event queue and the maximum number of elements the queue can hold.

Manipulating the Pointer

The following function calls support manipulating the mouse cursor, also called the pointer:

MouRemovePtr
: Defines a restricted screen area where the pointer cannot appear

MouDrawPtr
: Redefines a restricted screen area, allowing the pointer to appear

MouGetPtrPos/MouSetPtrPos
: Gets or sets the vertical and horizontal positions of the pointer

MouGetPtrShape/MouSetPtrShape
: Gets or sets the shape of the pointer.

Defining Collision Areas

An application synchronizes its use of the screen with the mouse device driver by issuing **MouRemovePtr** and **MouDrawPtr** calls.

Calling MouRemovePtr defines a rectangular pointer image restricted area, called the collision area, which is for exclusive use by the application and where the pointer cannot appear. When the first MouOpen of a session is issued and the mouse is initialized, the collision area is defined as the size of the screen.

Calling MouDrawPtr notifies the mouse device driver that an area previously restricted to application use by a MouRemovePtr call is now available to the mouse device driver for drawing the pointer image. If the pointer is positioned in the previously restricted area, its image is drawn.

Text-Windowed Considerations: Collision avoidance is provided automatically by the Presentation Manager when the pointer is within the default window. No operation is performed for MouRemovePtr, and a return code of 0 is returned.

The Presentation Manager system pointer is always displayed in a default window. No operation is performed for MouDrawPtr, and a return code of 0 is returned.

Sample Programs: Examples of the **MouRemovePtr** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSEC.C
\TOOLKT12\MASM\SAMPLES\BSE\MOUSAMPB\MOUSAMP.ASM
```

Setting the Pointer Position

MouGetPtrPos and **MouSetPtrPos** are used to determine and set the pointer position. The pointer's position is specified as row and column coordinates in either pel or character units, depending upon the current display mode of the session. Pel values are used for graphics modes and character units are specified for text modes.

MouSetPtrPos has no effect on a currently defined collision area.

Text-Windowed Considerations: Conceptually, the default window is a view on the usually larger logical video buffer. The position of the mouse pointer can be outside that view and yet still be within the extents of the logical video buffer. In this case, MouGetPtrPos returns the coordinates of the cell under the mouse pointer. If the pointer is outside the extents of the logical video buffer image, MouGetPtrPos returns the coordinates of the nearest logical video buffer position. In either case, the logical video buffer is scrolled when necessary, until the specified logical video buffer cell appears within the view window.

MouSetPtrPos sets the pointer position, but only under the following conditions; the position specified must be visible in the window, and the default window must be the active window. If these conditions are not met, the pointer position is not set, and an error is returned.

Sample Programs: Examples of the **MouSetPtrPos** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSEC.C
\TOOLKT12\MASM\SAMPLES\BSE\MOUSAMPB\MOUSAMP.ASM
```

Setting the Pointer Shape

MouSetPtrShape allows an application to define the pointer shape to the mouse device driver.

The pointer bit image to be used by the mouse device driver is stored in PtrBuffer.

PtrDefRec is the area where the application stores the necessary data for the pointer draw device driver to build a row-by-column image for each bit plane of the current display mode.

Issuing **MouGetPtrShape** returns a copy of the pointer shape being used by the mouse device driver for the current session.

An application can pass a parameter list with the same meaning as defined for MouSetPtrShape to MouGetPtrShape. The parameters that describe the pointer shape and attributes are copied into the pointer definition control block (pointed to by PtrDefRec) by the mouse device driver. The pointer definition record must contain the size of the application buffer where the device driver returns the pointer image.

The pointer shape you can define may be limited by the support provided by the display adapter installed in the user's system. For example, if the installed adapter supports only text modes 0, 1, 2, 3, or 7, and text mode is in effect, the pointer is supported only as a reverse video block, which allows characters already on the screen to remain visible. The reverse video block is the default. Its width and height are equal to 1.

The pointer shape is tracked by the pointer draw device driver.

Text-Windowed Considerations: Mouse support is limited to text mode emulation, and application-defined pointers for these modes are not displayed by the Presentation Manager. Instead, the Presentation Manager system default pointer is displayed in the default window.

Although the pointer shape defined with MouSetPtrShape has no meaning to the Presentation Manager, it is returned by a subsequent MouGetPtrShape. MouGetPtrShape works the same as in the base operating system with the limitation that the pointer shape is always the text-mode pointer defined by MouSetPtrShape.

Registering Another Mouse Subsystem

Like any of the OS/2 I/O subsystems, the mouse subsystem consists of subroutines that can be individually replaced.

The following function calls support registering an alternate mouse subsystem:

MouRegister
: Registers another mouse subsystem for the current session

MouDeRegister
: Cancels the registration of a mouse subsystem

MouSynch
: Synchronizes access for a mouse subsystem with the mouse device driver.

These calls cannot be used by a text-windowed application. (Of course, **none** of the mouse calls can be used by a Presentation Manager application.)

Replacing Mouse Function Calls

Individual replacement of OS/2 mouse function calls is accomplished by setting bits in a function call bit map specified with **MouRegister**. Once the alternate mouse subsystem is registered, and an OS/2 mouse call is issued, OS/2 checks the bit map for the call.

If a bit is set for the call, indicating a replacement for the function, the alternate dynamic link module is called, and control is passed to the entry point name of the routine specified with MouRegister.

MouRegister remains in effect within the session until a **MouDeRegister** is issued or the session ends. MouDeRegister must be called by the same process that issued MouRegister.

Synchronizing Mouse Access

MouSynch is intended for use by an alternate keyboard subsystem to synchronize access to the mouse device driver. An exclusive semaphore is used by MouSynch, which blocks other threads in the session until the semaphore is cleared.

Part 7. Control Program Functions

Chapter 32. Memory Management

This chapter describes how an application can take advantage of the memory management features of OS/2 to make efficient use of memory. Memory management exploits the segmentation capability of the protected address mode of the 80286 and 80386 Intel microprocessors.

The OS/2 1.2 Programming Tools do not provide sample programs for all function calls discussed in this chapter. However, when an example of a function call is used in a sample program, the subdirectory path of the sample program is shown. **Note that a sample program may not illustrate the task of the function call as discussed in the chapter text.**

Memory Management Design Considerations

The 80386 microprocessor contains the basic functions required to implement page-oriented virtual memory systems with page level protection. Each page is 4KB of contiguous physical memory. It is recommended that application developers design for the existing segmented system and the paged environment. Applications should use segments that are multiples of 4KB and refrain from allocating a large number of small segments.

Segments allocated in a paged environment may be mapped to physical memory *rounded up* to a multiple of 4KB. Applications with many small segments could use memory inefficiently in a paged environment. For instance, a 2 byte, 1KB, and 4KB segment could all use the same amount of physical memory, one 4KB page. In this environment a 13KB segment would take four 4KB pages, three which are full, and one which contains the last 1KB. If an application requires a large number of small pieces of memory, it is recommended that the application allocate a single large segment and use the OS/2 suballocation functions or application specific memory management to suballocate the single segment.

Segments should be as close to a multiple of 4KB without exceeding the 4KB boundary, but small enough (16KB to 32KB) to make efficient use of memory in a non-page environment. The LINK option, PACKCODE, may be useful in packing code segments into a number of physical segments each containing nb bytes.

Dynamic Data Segments

Memory management calls enable your application to allocate memory dynamically for the purpose of holding data. When a call is made to allocate memory, OS/2 builds a descriptor entry for the Local Descriptor Table (LDT) of the requesting process and returns a 16-bit selector that is used like a handle to access the allocated memory. The LDT contains descriptors that define the addresses of all the segments used by the process.

Creating and Accessing Private Segments

The following function calls support memory allocation:

DosAllocSeg
: Allocates a segment up to 64KB in size

DosAllocHuge
: Allocates multiple 64KB segments as one logical chunk of memory

DosGetHugeShift
: Computes selectors for segments allocated by DosAllocHuge

DosMemAvail
: Tests memory availability before allocation.

Allocating a Segment up to 64KB in Size

To allocate a segment of memory of up to 64KB in size, use the **DosAllocSeg** call and specify the segment size. The segment created can be moved and swapped by OS/2.

The system returns a selector. To access the segment, load the selector into a segment register.

Figure 32-1 on page 32-2 contains a C programming example of a call to DosAllocSeg, taken from the MEMORYC.C sample program. This sample program is found in the following subdirectory:

```
\TOOLKT12\C\SAMPLES\BSE\MEMORYCB
```

In this example, a 72-byte unshared segment is allocated, and a message is written to the screen.

When you issue DosAllocSeg, you can specify that the segment is shared with other processes. Another way to create a shared segment is by calling DosAllocShrSeg, which allocates a named shared segment of up to 64KB. See "Creating and Accessing Shared Segments" on page 32-3.

A segment allocated with DosAllocSeg also can be specified as discardable by the system when no longer needed. Discardable segments require calls that tell the system when memory is in use and cannot be discarded. These calls are discussed under "Creating and Accessing Discardable Segments" on page 32-4.

Sample Programs: Examples of the **DosAllocSeg** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\MEMORYCB\MEMORYC.C
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC1\GRAPHIC1.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\C\SAMPLES\PM\IMAGE\IMAGE.C
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB\MEMORY.ASM
```

Allocating Multiple Segments

Some processes need more than 64KB of memory to accommodate large data structures, such as spreadsheets and large files. You could use the DosAllocSeg call to create this memory, but this would require your maintaining an array of selectors, one for each segment of memory, and complex coding to cross a segment boundary.

The **DosAllocHuge** call allows a process to allocate a large amount of memory by telling the system how many 64KB segments it needs and what the size of the last segment will be. The system allocates this memory, which can be moved and swapped, and returns the selector to the first segment. When this selector is used with a call, the requested function is performed for the entire portion of memory.

Issuing **DosGetHugeShift** returns a shift count, used to derive the selectors that access the individual segments of DosAllocHuge memory. To obtain the selector for the next 64KB of memory, take the value 1 and shift it to the left by the number of bits in the shift count. Use the resulting value as the increment to add to the previous selector, using the selector returned by DosAllocHuge as the first selector.

Adding a constant value to obtain access to the next logical segment is similar to the way in which contiguous physical segments are accessed in the PC-DOS environment; by adding a constant value to the segment register. This similarity allows the DosAllocHuge function to be included as part of the family API.

When you issue DosAllocHuge, you can specify that the allocated memory is shared by other processes. You can also specify the memory as discardable by the system when no longer needed. See "Creating and Accessing Shared Segments" on page 32-3 and "Creating and Accessing Discardable Segments" on page 32-4.

Definitions

```
#define  NOT_SHARED     0          /* indicating that segment is not shared  */
#define  SIZE           72         /* size of the segment being allocated    */
```

Variables

```
   SEL segment_sel;                 /* returned selector placed here   */
```

The call to DosAllocSeg

```
   error_code = DosAllocSeg(SIZE, (unsigned far *) &segment_sel, NOT_SHARED);
   if (error_code != NO_ERROR) {
      printf("error code = %d \n",error_code);
   }
```

Figure 32-1. Allocating a Segment

Determining Available Memory

Applications should be discretionary in claiming large memory that can impair system performance. For example, the usability of an application such as a word processor depends on its capability to load a large file into memory. However, another type of application (such as a copy utility) does not need this capability to make it usable. Claiming large memory should be avoided when doing so can cause wholesale swapping out of segments in a low memory situation. When it is practical, an application should access the disk.

To avoid allocating large memory that affects system performance, you can take a snapshot of available system memory by issuing **DosMemAvail**. This call returns the size of the largest block of unallocated memory. Although this value can change at any time because of system activity, it can provide a good indication of the system memory state.

Creating and Accessing Shared Segments

Allocating memory that is shared by other processes is the cornerstone of interprocess communication. For a description of the mechanisms that enable processes to coordinate sharing of memory, see Chapter 35, "Communications Between Threads and Processes" on page 35-1.

Allocated memory can be shared locally within a process, or it can be given a name and shared globally by any process in the system that knows the name.

Allocating a Locally Shared Segment

The following function calls support local sharing of memory between processes:

DosAllocSeg/DosAllocHuge
: Allocates the memory as shared

DosGiveSeg
: Allows a child process to share memory

DosGetSeg
: Accesses the shared memory.

After a process has allocated a segment or multiple segments and designated the memory as shared, the process can then call DosGiveSeg to allow a child process to share the segment. The process passes the segment's selector and the process ID of the intended sharer to the system.

Note: A process ID is obtained by a parent process when it executes a child process with DosExecPgm. For a description of DosExecPgm, see "Starting Another Process" on page 34-9.

The system responds to the calling process by returning a selector for the sharer to use when accessing the segment with DosGetSeg. The caller can pass this selector to the sharer by means of a queue. See "Passing Addresses to Data with a Queue" on page 35-8.

DosGetSeg will not lock discardable segments. If the sharer wants a shared discardable segment locked, it must call DosLockSeg. See "Creating and Accessing Discardable Segments" on page 32-4.

Allocating a Globally Shared Segment

The following function calls support global sharing of named segments:

DosAllocShrSeg
: Allocates a named shared segment

DosGetShrSeg
: Accesses the named shared segment.

Issuing DosAllocShrSeg creates a segment, up to 64KB in size, that can be moved and swapped by OS/2 and has a symbolic name. This segment can be accessed by any process in the system that knows its name. To access the segment, the process specifies the segment name with DosGetShrSeg, which returns the segment's selector to the process. This selector is the same selector returned when the shared segment was created.

The segment name specified with DosAllocShrSeg can be any valid OS/2 file name, but because the name has to go in the \SHAREMEM subdirectory, it must start with \SHAREMEM\. The maximum number of named shared segments a process can define is 10.

When DosGetShrSeg is issued, a usage count for the segment is incremented. The usage count is decremented when **DosFreeSeg** is issued.

Accessing A Data Segment as a Code Segment

A process can call a data segment as a code segment by creating a code segment selector for it with **DosCreateCSAlias.** When DosCreateCSAlias is issued with a data segment selector, the system creates for its own use a code segment descriptor for the segment in the Local Descriptor Table of the requesting process. It returns a code segment selector the process can use in CS to execute code contained in the segment.

To qualify for use by DosCreateCSAlias, the selector must access a writable data segment that is exclusively owned by a process. This segment cannot be part of large memory allocated by a DosAllocHuge call. This means that selectors for shared segments, resource segments, and dynamic link library global data segments cannot be used. Any selector whose data segment meets these criteria and is valid for the DS, ES, or SS registers can be used.

After the data segment selector is specified with a DosFreeSeg call, the segment is no longer accessible as a data segment. However, the code in the segment can still be referenced with the code segment selector. Another DosFreeSeg call is required to free the code segment selector.

Note: Code in the segment can be called from privilege level 3 or I/O privilege level.

Creating and Accessing Discardable Segments

It is good practice to allocate memory as discardable when it is used to hold objects that are accessed for short periods of time and can be easily regenerated. Examples of such objects are cache buffers for a data base application, saved bit map images for obscured windows, and precomputed display images for a word processing application.

The following function calls support creating and accessing discardable memory:

DosAllocSeg, DosAllocHuge
Allocates memory as discardable

DosLockSeg
Locks discardable memory

DosUnlockSeg
Unlocks discardable memory

DosReallocSeg, DosReallocHuge
Reallocates a discardable segment.

A process accesses memory that has been designated as discardable with DosAllocSeg or DosAllocHuge by issuing DosLockSeg. This call tells the system that discardable memory is in use and cannot be discarded; however, the system can still move and swap the memory. When the process has finished using the memory, the process unlocks it with DosUnlockSeg, allowing the system to discard the segment and reclaim the space in a low memory situation.

If memory has been discarded, a DosLockSeg call to it causes an error message. The memory must then be reallocated by issuing **DosReallocSeg** or **DosReallocHuge** and the data must be regenerated. Reallocating the memory also locks it. To unlock discardable memory locked by reallocation, issue DosUnlockSeg.

To gain access to a discardable segment that is also shared use DosGetSeg. DosLockSeg locks the shared segment. To release the shared segment for discarding, the sharer issues DosUnlockSeg. When a process has already locked a shared segment, another process that wants to use the segment can make multiple attempts to lock it. However, if multiple attempts are made to lock a segment, then the same number of calls must be made to unlock it.

Note: It is an error to unlock a segment that is already fully unlocked.

Once a segment has been locked 255 times, it becomes permanent in memory. Additional calls to DosLockSeg and DosUnlockSeg have no effect on the segment's locked state.

Remember, DosAllocSeg, with AllocFlags bit 2 (01100B) set, or DosReallocSeg, will allocate memory and perform the same action as a call to DosLockSeg. DosLockSeg and DosUnlockSeg have no effect on segments which are non-discardable.

Suballocating Pieces of a Segment

There are times when a process needs only small amounts of memory. The following Memory Suballocation Package calls can be used by a process to carve out pieces of memory from a previously allocated segment:

DosSubSet
Initializes a segment for suballocation

DosSubAlloc
Suballocates amounts of memory within the segment.

The Memory Suballocation Package keeps track of suballocated memory. Suballocating memory is faster than allocating individual segments, because the processor is not involved. However, this also means that individual suballocated pieces are not protected.

To initialize a segment for suballocation, issue **DosSubSet**, specifying the segment's selector returned by a previous DosAllocSeg or DosShrSeg call. Set the flag to 1 to indicate initialization. The size of suballocated memory should be a multiple of 4 bytes. If it is not a multiple of 4, DosSubSet *rounds it up* to a multiple of 4, regardless of the actual size of the segment at the time of allocation. A size parameter of zero indicates that the segment is 64KB. The minimum rounded size that can be set is twelve.

Issue **DosSubAlloc** to suballocate the memory within the segment. Although OS/2 can move or swap a segment used for suballocation, it does not move or swap the suballocated pieces individually. The amount of memory lost to fragmentation in a segment used for suballocation can be reduced by suballocating memory blocks whose size is a multiple of 4.

If a size specified for suballocated memory is not a multiple of 4 bytes, DosSubAlloc or DosSubFree *rounds it up* to a multiple of 4. No paragraph alignment is required because all requests are serviced on a byte alignment basis. A size parameter of zero is an error. The system returns the offset of the suballocated memory.

Figure 32-2 on page 32-6 contains C source code from the MEMORY.C sample program that illustrates suballocating memory. For Macro Assembler coding examples of the calls used to suballocate a segment, refer to the MEMORY.ASM sample program in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB
```

Changing the Size of Allocated Memory

Previously allocated memory is reallocated after it has been discarded by the system or when its size needs to be increased or decreased.

The following function calls support reallocation of memory:

DosReallocSeg
Reallocates a segment up to 64KB in size

DosReallocHuge
Reallocates multiple 64KB segments

DosSubSet
Resets a reallocated segment for suballocation.

Reallocating a Segment up to 64KB

DosReallocSeg allows a process to make more efficient use of both private and shared segments by increasing and decreasing their size, depending on the needs of the process.

If the segment has been discarded, issuing DosReallocSeg has the combined effect of an allocation call and a DosLockSeg call; the reallocated segment is locked in memory. See "Creating and Accessing Discardable Segments" on page 32-4.

Figure 32-3 contains a coding example from the MEMORYC.C sample program that illustrates changing a segment's size to 64KB. Note that the value 0 represents the default of 64KB.

```
error_code = DosReallocSeg(0, segment_sel);
if (error_code != NO_ERROR) {
   printf("error code = %d \n",error_code);
}
```

Figure 32-3. Changing a Segment's Size

Reallocating Multiple Segments

DosReallocHuge changes the size of both private and shared memory originally allocated with DosAllocHuge. The new size can be either smaller or larger than the original size, as long as it does not exceed the amount specified for the maximum number of segments in the DosAllocHuge call.

Definitions

```
#define  NOT_SHARED     0        /* Indicates segment is not shared      */
#define  SIZE           72       /* Size of the segment being allocated  */
#define  SIZE2          0        /* Size defaults to size of 64KB        */
#define  SUBSIZE        32       /* Size of the segment suballocated     */
#define  SUBSIZE2       64       /* Size of the segment suballocated     */
```

Variables

```
   SEL segment_sel;              /* Returned selector placed here        */
   unsigned result = 0;          /* Contains the result code upon exit   */
   unsigned offset,offset2;      /* Hold the offset values returned      */
   unsigned init = 1;            /* Parameter set to initialize segment  */
```

Allocating an unshared segment

```
   error_code = DosAllocSeg(SIZE, (unsigned far *) &segment_sel, NOT_SHARED);
   if (error_code != NO_ERROR) {
      printf("error code = %d \n",error_code);
   }
```

Initializing the segment for suballocation

```
   error_code = DosSubSet(segment_sel, init, SIZE);
   if (error_code != NO_ERROR) {
      printf("error code = %d \n",error_code);
   }
```

Suballocating a 32-byte block and outputting a message

```
   error_code = DosSubAlloc(segment_sel, (unsigned far *) &offset, SUBSIZE);
   if (error_code != NO_ERROR) {
      printf("error code = %d \n",error_code);
   }
```

Freeing a 64-byte suballocated block and outputting a message

```
   error_code = DosSubFree(segment_sel, offset, SUBSIZE2);
   if (error_code != NO_ERROR) {
      printf("error in subfree, error code = %d\n", error_code);
   }
```

Freeing the entire segment

```
   error_code = DosFreeSeg(segment_sel);
   if (error_code != NO_ERROR) {
      printf("error code = %d \n",error_code);
   }
```

Figure 32-2. Suballocating Memory within a Segment

Reallocating a Suballocated Segment

Changing the size of a suballocated segment requires two calls: DosReallocSeg to reallocate the memory, and DosSubSet to reset the segment for suballocation.

Call DosReallocSeg and specify the new segment size in bytes and the selector of the original segment. Then call DosSubSet and specify the selector, set the flag to 0, and specify the new segment size in bytes. Setting the flag to 0 indicates the suballocated segment's size has been changed.

Neglecting to issue DosSubSet can cause unpredictable results when suballocation is performed on the segment.

Sample Programs: Examples of the **DosReallocSeg** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\GRAPHIC1\GRAPHIC1.C
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB\MEMORY.ASM
```

Freeing Memory

The following calls support memory deallocation:

DosFreeSeg
: Frees segments allocated with DosAllocSeg, DosAllocShrSeg, and DosAllocHuge

DosSubFree
: Frees a block of memory suballocated with DosSubAlloc.

To free memory segments that have been allocated with DosAllocSeg, DosAllocShrSeg, and DosAllocHuge, issue DosFreeSeg with the selector of the memory to be freed.

In the case of DosAllocHuge, the selector used is the initial selector returned by DosAllocHuge. Issuing DosFreeSeg with this selector frees the entire block of memory allocated by DosAllocHuge.

When memory is shared, either locally or globally, a usage count is maintained. Each time a process accesses a globally shared segment using DosGetShrSeg or gives a locally shared segment with DosGiveSeg to another process, the usage count is incremented. DosFreeSeg decrements the usage count. When the usage count is 0, the shared segment is freed.

DosFreeSeg is also used to free segments specified with DosCreateCSAlias and resource segments created for Presentation Manager applications. See "Accessing A Data Segment as a Code Segment" on page 32-4 and "Creating Resource Segments for Presentation Manager Applications."

To free a block of suballocated memory, issue **DosSubFree** and specify the segment selector and the offset of memory to be freed, as well as its size. The size parameter should be a multiple of four bytes. If it is not a multiple of 4, **DosSubFree** rounds it up to a multiple of 4 bytes. A size parameter of zero is an error.

Sample Programs: Examples of the **DosFreeSeg** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\MEMORYCB\MEMORYC.C
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC1\GRAPHIC1.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\C\SAMPLES\PM\IMAGE\IMAGE.C
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB\MEMORY.ASM
```

Examples of the **DosSubAlloc**, **DosSubFree**, and **DosSubSet** calls can be found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\MEMORYCB\MEMORYC.C
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB\MEMORY.ASM
```

Obtaining the Size of Allocated Memory

Issuing **DosSizeSeg** returns the size of an allocated segment. This function call is provided for family applications, which are designed to run in the PC-DOS environment, as well as the multitasking environment of OS/2.

If huge memory is the target of this call, the base selector must be used. DosSizeSeg will fail on non-base huge subsegments.

Creating Resource Segments for Presentation Manager Applications

Resources for a Presentation Manager application, such as symbols and bit maps, can be compiled by the resource compiler as part of a dynamic link module or as part of the application's executable file. Having resources in the executable file means that a single file contains all of the code and data for an application.

Read-only data segments containing resources can be accessed dynamically at run time. These segments are identified as PRELOAD or LOADONCALL segments in the module definition file (see "Dynamic Linking" on page 34-15).

Resource Design Considerations

OS/2 has added new functions **DosGetResource2** and **DosFreeResource** to load and free an application specific resource. DosGetResource2 returns a far pointer to a resource rather than the selector returned via **DosGetResource**. Applications that wish to execute on OS/2 1.1 and 1.2 should use the

OS/2 runtime dynamic link capabilities, DosLoadModule and DosGetProcAddr, to get the address of DosGetResource2 and DosFreeResource. If the DosGetProcAddr call to obtain the address of DosGetResource2 and DosFreeResource fails, the application can call DosGetResource and DosFreeSeg. Applications that use DosGetResource2 and DosFreeResource allow OS/2 to optimize memory allocation associated with the application's resource.

Note: Although DosGetResource and DosFreeSeg remain viable calls, it is recommended that applications targetted for OS/2 Version 1.2 use DosGetResource2 and DosFreeResource.

Accessing a Read-Only Resource

DosGetResource2 obtains a far pointer to a read-only resource. The access key specified with DosGetResource2 comprises two 16-bit numbers assigned as Type ID and Name ID at the time of its creation. These numbers are viewed similar to a filename and a filename extension.

A handle is also specified with DosGetResource2 to indicate the source of the resource. If 0 is specified, the resource is in the executable file of the current process. If the handle is a non-zero number, the resource is in a dynamic link module, and the handle is the one returned when the dynamic link module was loaded with DosLoadModule.

Freeing a Resource

To free a resource that is defined as LOADONCALL, **DosFreeResource** must be issued with the resource's far pointer as many times during the life of the process as DosGetResource2 was called. This prevents one thread of a process from freeing a resource that is currently being used by another thread of the same process. After the last reference to a resource is freed, the memory becomes available for reuse by the system. However, the memory is not reused until the system determines it cannot satisfy a memory allocation request. This allows the resource to remain in memory in case the process issues another DosGetResource2 call. The system thus avoids having to reread the contents of the resource from disk.

To free a resource that is defined as PRELOAD, DosFreeResource must be called by the process one more time than DosGetResource2 is called. This is because at load time the PRELOAD resource is treated as if an implicit DosGetResource2 is performed.

Chapter 33. File and Disk Management

This chapter describes OS/2 file system function calls. This chapter also describes the new OS/2 installable file system mechanism, including:

- The High Performance File System
- Long file names
- Extended attributes.

The OS/2 1.2 Programming Tools do not provide sample programs for all function calls discussed in this chapter. However, when an example of a function call is used in a sample program, the subdirectory path of the sample program is shown. **Note that a sample program may not illustrate the task of the function call as discussed in the chapter text.**

Using OS/2 File Systems

Applications primarily use OS/2 file systems to manage storage devices. File systems, for example, manage block devices such as fixed disks. A file system allows applications to create and manage file objects on storage devices and to perform I/O to those file objects. Applications, for example, make files and subdirectories and read from and write to files. Whereas an application views files as logical sequences of data, a file system manages the physical locations of data on the storage device, for the application.

Applications may also use OS/2 file system function calls to perform I/O to:

- Character devices.

 These include the COM, clock, console (keyboard and screen), screen, keyboard, printer, nul, pointer and mouse devices.

- Standard devices.

 These are character devices that are automatically installed by OS/2 and recognized by the file system as the Standard INPUT, Standard OUTPUT, and Standard ERROR devices.

- Pseudo-character devices.

 These are single file devices which may be redirected. The behavior of one of these files is very similar to the behavior of a normal DOS character device. It may be read from by calling DosRead and written to by calling DosWrite. However, DosChgFilePtr and DosFileLocks functions can also be applied to the file. The user perceives this file as a device name for a non-existing device. This file is seen as a character device because the current drive and directory have no effect on the name. A pseudo-character device name consists of an ASCII string containing the name of an OS/2 file in a subdirectory called \DEV\.

- Logical file devices.

 These are the fixed disk/diskette drives or the partitions on the fixed disk/diskette drives.

To perform I/O to a file or device, an application must first establish a connection to the new or existing file or device. This is done by calling DosOpen, which returns a handle to the file or device. A handle is a 16-bit identifier associated with the file or device and any information associated with the file or device. An application then performs I/O to the file or device by using the handle in subsequent file system function calls. When I/O has been completed, the application terminates the connection to the file or device by calling DosClose, which frees the handle to the file or device. For example, to read data from a file, an application includes instructions for the following:

```
CALL DosOpen to get a handle to the file
CALL DosRead to read data from the file
CALL DosClose to free the handle to the file
```

Similarly, to perform I/O to a pipe, an application must first establish a connection to the pipe. To gain read and write access to an unnamed pipe, an application calls DosMakePipe. DosMakePipe returns a handle to the unnamed pipe. To gain read and write access to a named pipe, an application must first define the pipe by calling DosMakeNmPipe. DosMakeNmPipe returns a handle to the named pipe. The application that defines a named pipe by calling DosMakeNmPipe is the owner of the named pipe. Other applications that wish to perform I/O to the named pipe are called clients. Clients of a named pipe gain read and write access to it by calling DosOpen. For more information on pipes see "Two-Way Communication with Named Pipes" on page 35-11.

File system function calls that support I/O for character devices are:

- DosOpen
- DosRead
- DosWrite
- DosDupHandle
- DosClose
- DosReadAsync
- DosWriteAsync.

By convention, the following file system handles have special meaning:

Handle	Meaning
0	Standard INPUT Device
1	Standard OUTPUT Device
2	Standard ERROR Device

Installing OS/2 File Systems

OS/2 Version 1.2 supports the coexistence of multiple, active file systems in a single PC. These include:

- the OS/2 FAT file system
- OS/2 installable file systems (IFS).

Note: The OS/2 FAT file system is based on the DOS FAT file system. FAT stands for "File Allocation Table" and refers to the data structures used to track file objects and disk space. The file allocation table is a table of linked lists of sectors on a disk. A file, for example, may reside on many sectors on a disk and is represented in the file allocation table as a linked list of those sectors. A FAT file system accesses data in a file by locating the file's sectors in the file allocation table.

OS/2 file systems are accessible to applications through a standard set of function calls.

OS/2 Version 1.2 FAT and installable file systems both support:

- Existing logical file and directory structure
- Existing naming conventions
- Multiple logical volumes (partitions)
- Multiple and different storage devices
- Redirection or connection to remote file systems
- Extended attributes (EA)
- Meta (global) character processing.

In addition, OS/2 Version 1.2 installable file systems support:

- Long file names
- An extendable application interface.

OS/2 file systems manage file I/O and control the format of information on the storage media. Installable file systems are implemented as file system drivers, or FSDs. A file system driver is loaded during system initialization when an IFS= statement is encountered in the CONFIG.SYS file. The FAT file system is loaded automatically during system initialization. It does not require an IFS= statement in CONFIG.SYS.

A storage device is managed by only one file system at a time. When OS/2 is started, storage devices are assigned drive letters. A physical storage device may be divided into multiple partitions or volumes (that is, logical storage devices), each of which is referenced by a different drive letter. See the description of the FDISK utility in the OS/2 Version 1.2 Online Command Reference for details on partitioning a storage device.

Each logical storage device may be managed by a different file system. The file system that is attached to a storage device manages that device. A user attaches a file system to a storage device by:

- Loading the file system driver during system initialization (that is, including an IFS= statement in CONFIG.SYS).
- Formatting the storage device using the format options for the file system.

The file system that is used to format the storage media manages that media each time the system is started, as long as it is loaded during system start-up. OS/2 directs file system requests for a storage media to the file system that recognizes the format of the media, that is, to the file system that formatted the media. When no file system recognizes the format of the media, the OS/2 FAT file system attempts to manage the media. This may occur when the file system that is used to format the storage media is not loaded during system start-up; that is, the IFS= statement has been removed from the CONFIG.SYS file after OS/2 installation. If the OS/2 FAT file system is then unable to recognize the media format (the media may have a different directory structure), the user is returned an error when attempting to access the media.

For Version 1.2, users have the option during installation of OS/2 to format their fixed disks using the High Performance File System (HPFS), an installable file system. When this option is selected, an IFS= statement is added to CONFIG.SYS so that the High Performance File System is automatically loaded during each system start-up.

Formatting associates the FSD with the logical storage device or drive letter. When an application makes a file system function call, OS/2 Version 1.2 directs the request to the file system managing the storage device:

- To an installable file system
- To the FAT file system, if no installable file system is loaded and attached to the storage device.

For example, a system is configured with diskette drives A and B and fixed drives C and D. During OS/2 installation, the user selects the option to format fixed disk C using the High Performance File System. Fixed disk C is, then, managed by HPFS. Fixed disk D, formatted using a previous version of OS/2, is managed by the OS/2 FAT file system. Diskette drives A and B are managed by the OS/2 FAT file system. (Removable media may not be formatted using HPFS.)

When an application calls DosOpen to open a file on C, OS/2 directs the request to the High Performance File System. When an application calls DosOpen to open a file on storage device A, OS/2 directs the request to the OS/2 FAT file system. See Figure 33-1.

In the example shown in Figure 33-1, if HPFS is not loaded during system start-up, the OS/2 FAT file system receives file system requests for fixed disk C. However, because HPFS supports a different directory structure than FAT, the OS/2 FAT file system cannot recognize file objects on the disk. The user is then returned an error when attempting to access the disk.

The user can determine which file system was used to format a storage device by using the OS/2 Version 1.2 CHKDSK utility. CHKDSK displays a message indicating which file system manages the specified drive. For more information on FORMAT and CHKDSK, see the OS/2 Version 1.2 Online Command Reference.

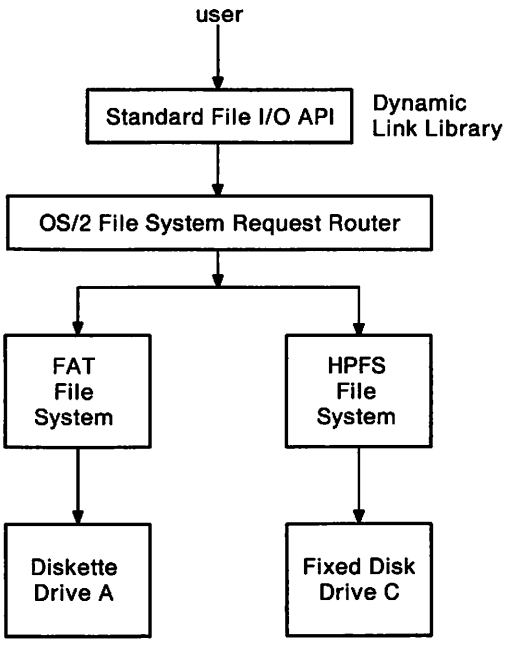


Figure 33-1. OS/2 Version 1.2 handling of file system requests

Recognizing DOS and OS/2 File Objects

Figure 33-2 on page 33-4 illustrates those disks that can be accessed by applications running under DOS and under OS/2.

For compatibility reasons, OS/2 requires that the OS/2 FAT file system recognizes file objects created by the DOS FAT file system. This means that applications running under OS/2 (these include both OS/2 mode and DOS mode applications) can recognize file objects created by applications running under DOS. OS/2 recognizes files and directories on DOS disks.

Because the OS/2 FAT file system supports the same directory structure as the DOS FAT file system, the DOS FAT file system can recognize file objects created by the OS/2 FAT file system. This means that applications running under DOS can recognize files and directories on OS/2 FAT disks.

However, HPFS does not support the same directory structure as the DOS FAT file system. The DOS FAT file system cannot recognize file objects created by the OS/2 HPFS. This means that applications running under DOS cannot recognize files and directories on OS/2 HPFS disks.

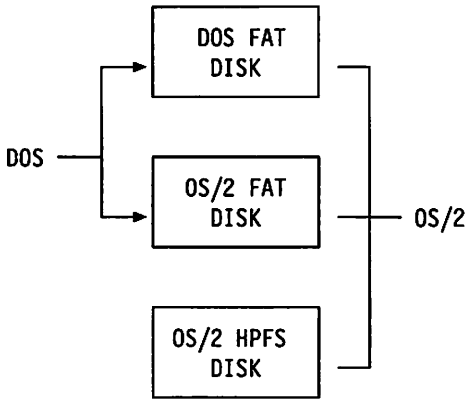


Figure 33-2. Accessing DOS and OS/2 disks

Because of the manner in which OS/2 handles file system requests (see Figure 33-1), DOS mode applications running under OS/2 can recognize files and directories on FAT and HPFS disks. For example, a DOS mode request to read a file located on a FAT disk is handled by the OS/2 FAT file system. Similarly, a DOS mode request to read a file located on an HPFS disk is handled by HPFS.

Using the High Performance File System

The High Performance File System (HPFS) provides extremely fast access to very large disk volumes. Its features include:

- Caching of directories, data, and file system structures
- Multi-threaded I/O
- Write-behind logic
- Optional write-through
- Strategic allocation of directory structures
- Highly contiguous file allocation
- Enhanced recoverability
- Extended attribute support
- Long file name support
- Bootability.

Each element of a full path name residing on an HPFS disk may consist of up to 254 characters. For example, "\xxx...xxx\" is an element in a full path name. "xxx...xxx" may include up to 254 characters. For more information on long file name support by installable file systems see "Using Long File Names" on page 33-8.

HPFS manages a cache of memory divided into blocks of 2KB. Data that is read from and written to the disk is transferred through this cache. When a user request specifies data that is not present in the cache, HPFS selects the least recently used (LRU) block, writes the data within that block to disk if necessary, and then fills the block with the requested data. This significantly increases the chances that often-used data is found in memory, thereby saving the expense of a disk-read operation.

In most cases, when a write-data request is received, it is not necessary to write the data immediately to the disk. HPFS takes such data and copies it into the block cache without actually performing the disk-write operation. When the data is in the cache, it is written to disk as a background activity. This is referred to as "lazy writing" and allows the typical user-write operation to occur much faster than in file systems where all writes are synchronous. See Figure 33-3.

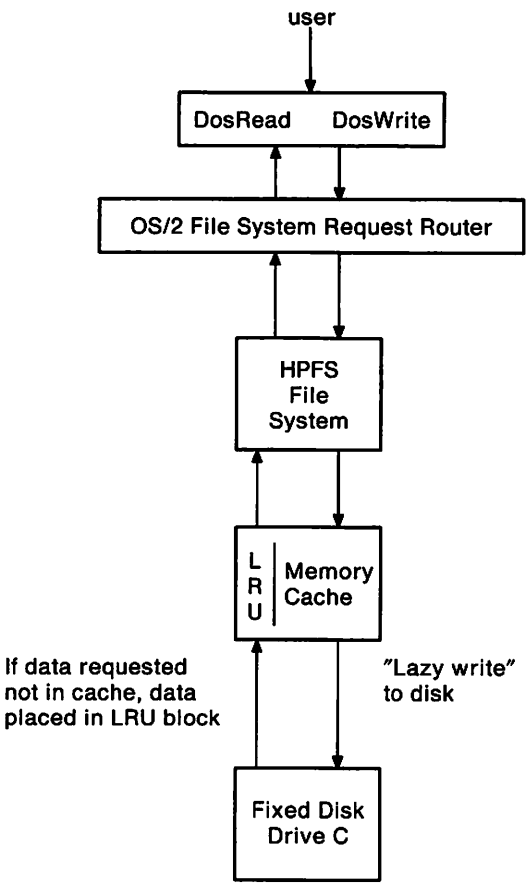


Figure 33-3. HPFS data caching

As stated previously, users have the option during installation of OS/2 Version 1.2 to install a bootable version of HPFS on their fixed disk. HPFS consists of:

- The HPFS file system driver, HPFS.IFS
- The HPFS lazy-write utility, CACHE.EXE
- The HPFS file system utilities, UHPFS.DLL.

The user determines the amount of lazy-write support by setting the following parameters on the command line invoking the lazy-write utility CACHE.EXE:

- MaxAge

 When the data in a cache block is more than MaxAge number of milliseconds out of date with respect to the disk, the block is queued for writing to the disk. This reduces the amount of data lost due to inopportune system shutdowns.

- DiskIdle and BufferIdle

 When no user I/O (non-lazy-write) request has been made for DiskIdle number of milliseconds, all cache blocks (in oldest-first order) that have not been touched for BufferIdle number of milliseconds are queued for writing to disk. This allows HPFS to write out user data during times of relative disk inactivity and to reduce the need for rewriting heavily used cached blocks.

Utilities for each file system are contained in a single dynamic link library. OS/2 invokes the utilities based on the file system that recognizes the volume on which the utility is to be run. The utilities that are included in the dynamic link library for each file system are:

FORMAT	Disk formatter
CHKDSK	File system validation and repair
RECOVER	File recovery
SYS	System installation.

HPFS disks, or partitions, can be created as bootable or non-bootable partitions. That is OS/2 may or may not be booted from a disk that was formatted using HPFS. A bootable HPFS disk is created during OS/2 installation when the user selects the option to install HPFS in a system partition. A non-bootable HPFS disk is created by the user after starting, or booting, the system from a FAT or HPFS partition.

The OS/2 Dual Boot function allows the user to start DOS from within OS/2 or OS/2 from within DOS. Both systems reside within the same partition. Because DOS does not recognize HPFS disks, this function is supported for FAT-based system partitions only.

Note: Multiple FAT and HPFS partitions may reside on the same physical disk in extended partitions. When multiple FAT and HPFS partitions reside on the same physical disk, any FAT partition that follows HPFS partitions cannot be accessed when booting from DOS. DOS stops at the first partition that it does not recognize.

For more information on the installation of bootable and non-bootable HPFS partitions, the installation of OS/2 dual-boot systems, and the invocation of the HPFS utilities, see *OS/2 Version 1.2 Getting Started*, *OS/2 Version 1.2 Advanced Features*, and the OS/2 Version 1.2 Online Command Reference.

Managing OS/2 File Systems

The following function calls allow applications to manage OS/2 file systems:

DosFSAttach
: Attaches or detaches a file system to/from a device

DosQFSAttach
: Gets information about a device and the file system that manages it

DosFSCtl
: Allows an application to communicate with a file system

DosShutDown
: Prepares a file system for system shutdown.

Attaching and Detaching File Systems

An FSD which uses a block device driver to do I/O to a local or remote (virtual disk) device is called a local file system. An FSD which accesses a remote system without a block device driver is called a remote file system.

An application, typically a network application, may call **DosFSAttach** to:

- Attach a drive to a remote file system
- Detach a drive from a remote file system
- Attach a pseudo-character device name to a local or remote file system
- Detach a pseudo-character device name from a local or remote file system.

This call establishes or destroys the connection between a drive or device and a file system. If an attach is successful, all requests to that drive or name are routed to the specified file system. If a detach is successful, the drive or name is no longer recognized by OS/2 in a file system call.

DosFSAttach does not support:

- Redirection of drive letters representing local drives
- Attachment to drives or devices that are not in the system's name space.

DosFSCtl can be used to attach to drives or devices not in the system's name space.

Getting Information About a Device and an FSD

DosQFSAttach allows an application to obtain information about all block devices and all character and pseudo-character devices, including:

- The type of device
- The name of the FSD to which the device is attached.

This information can be used to determine if OS/2 recognizes that a particular file system is attached to a storage device. This is important to an application that must guarantee that a particular file system is attached to a storage device. An application of this type must handle the situation where the FSD that formatted a certain disk has not been loaded during system start-up. (The user may have omitted the IFS= statement in CONFIG.SYS.) In such a case, there is a potential for the data on the disk to be destroyed because the wrong file system is attached to the disk by default.

Communicating With a File System

An extendable standard interface between an application and an FSD is provided by the **DosFSCtl** function call. This call is similar to the DosDevIOCtl function that provides a standard interface between an application and a device driver. An application sends a request to the FSD by specifying a particular FunctionCode. Data is exchanged through Data Areas and Parameter Lists.

DosFSCtl can be used to establish open connections to FSDs that are not attached to a name in the system's name space.

Preparing File Systems For System Shutdown

At any time during normal system operation, data destined for a disk may be in a cache. If this information is not written to disk before the system is restarted or powered-off, it will most likely result in the disk being corrupted. To prevent this from occurring, an application should call **DosShutDown** to make the system flush everything to disk and to prevent any further data being cached. The user can then safely power-off or reboot the machine.

Note: This function prepares all file systems and device drivers for system shutdown. This means that it should be called only when system shutdown is about to occur. The user and his applications will no longer be able to access their storage devices.

Naming OS/2 File Objects

All file systems have specific rules for constructing names of file objects. Different file systems may have different rules for naming file objects. The OS/2 FAT file system supports the DOS naming conventions. The OS/2 HPFS supports a superset of the DOS naming conventions, allowing for long file names and characters illegal under DOS. Although different file systems may have different rules for naming file objects, all OS/2 file systems require that full path names consist of directory and file names separated by backslashes (\).

For file objects managed by the FAT file system, the following rules apply:

- File names are limited to 8 bytes before and three bytes after a single dot. This is referred to as the 8.3 file name format.

 The 8-characters before the dot are blank-filled, with embedded blanks significant and trailing blanks not significant. For example, "file.a" is really "file .a ". "file.a" and "file .a " are treated as the same file by OS/2 and refer to the same file on disk. Also, "file.txt " and "file.txt" are treated as the same file by OS/2. Trailing blanks are truncated by OS/2 and any number of blanks immediately preceding the dot are not significant.

 However, blanks elsewhere in the name are significant: "f i l e.txt" is not the same as "file.txt".

- Names are case insensitive. This means that "file.TXT" and "File.Txt" refer to the same file. Lower and upper case characters are folded together for name comparison purposes.

- Names returned by file system function calls are uppercased. This means that if "file.TXT" is created, DosFindFirst returns "FILE.TXT".

- The invalid characters for file names are the range 0-1Fh and the characters

 < > + = ; , . " / \ []

 This list also applies to volume labels.

OS/2 Version 1.2 views path names as ASCII strings and no longer restricts file systems to the DOS file name format. Compatibility with existing DOS applications requires that all installable file systems support a superset of the FAT file system's 8.3 file name format.

For file objects managed by OS/2 installable file systems, the following rules apply:

- Each element of a full path name residing on a disk managed by an installable file system may consist of up to 255 characters. For example, in the path name "c:\xxx...xxx\xxy", "xxx...xxx" may include up to 255 characters. This is referred to as non-8.3 file name format, or long file names.

 Note: HPFS allows a maximum of 254 characters in an element of a full path name.

- Names are case insensitive.

- File name case as specified at create time is preserved. This means that if the file "file.TXT" is created, DosFindFirst returns "file.TXT". File name case may be modified using DosMove.

- Blanks immediately preceding a dot are significant. This means that "file.txt" and "file .txt" refer to different files.

- Trailing blanks are truncated. This means that "file.txt " is the same as "file.txt".

- Blanks elsewhere in the name are significant. This means that "f i l e.txt" is not the same as "file.txt".

- For compatibility reasons, trailing dots on component names are discarded. For example, "\foo.bar...bletch...\a..b...\c." becomes "\foo.bar...bletch\a..b\c". This processing includes semaphore, queue, pipe, module, shared memory names, and device names.

- The set of legal characters is expanded to include

 + = ; , []

 as well as all characters legal for the FAT file system.

- If an installable file system uses a component separator within a file name, it must be a dot (.). There are no restrictions on the number of components which may be allowed within a file name.

Creating Long File Names

When creating names for directories and files or when processing names supplied by the user, programs should follow these general rules:

- Process a path name as a null-terminated string. A program can determine maximum length for a path by using the DosQSysInfo function.

- Use any character in the current code page for a name, but do not use a path separator, a character in the range 0 through 31, or any character explicitly disallowed by the file system. Although a name can contain characters above 127, a program must be prepared to switch code pages if necessary to access the corresponding file. Compare names using a case-insensitive comparison. Names such as "ABC", "Abc", and "abc" are considered to be the same name.

- Use the backslash (\) and/or the forward slash (/) to separate components in a path name. No other character is accepted as a path separator.
- Use the dot (.) as a directory component in a path name to represent the current directory.
- Use the two-dot (..) as a directory component in a path name to represent the parent of the current directory.
- Use a dot (.) to separate components in a directory or file name. Unless explicitly defined by a file system, there are no restrictions on the number of components in a name.

Using Long File Names

For files located on file devices managed by the OS/2 FAT file system, long file names are handled differently in OS/2 mode than in DOS mode. In OS/2 mode, the name is considered an error. In DOS mode, the name is truncated and is not an error. The DOS mode treatment of file name formats provides compatibility with the PC-DOS environment for applications originally written for PC-DOS. However, if you are writing a family application to run in OS/2 and the PC-DOS environment, your application must allow for this difference in operating environments.

Programs that recognize long file names must indicate this by including the NEWFILES statement in their module definition file.

This statement directs the linker to set a bit in the executable file header. It indicates that the module supports long file names. This bit is meaningless in the DOS mode and on previous versions of OS/2. Programs written for OS/2 Version 1.2 installable file systems should set this bit. Bound programs that have this bit set see files with long file names in OS/2 mode and only files with 8.3 file name format in DOS mode.

This bit has meaning when attached to program modules, not when attached to DLLs. Whether the program recognizes long file names format is entirely dependent on the value of its NEWFILES bit and the bit's effect extends into calls to any DLLs. In order to be compatible with all OS/2 file systems, dynamic link libraries must not create internal temporary files or directories that do not comply with 8.3 file naming conventions. In addition, dynamic link libraries may not return long file names to an application. (The caller may be running on an 8.3 file name-only file system and use the returned name to create a file.)

OS/2 mode programs which do not recognize long file names can run under OS/2 Version 1.2 with some restrictions. See Figure 33-4 on page 33-9. These programs include programs written for previous versions of OS/2, as well as for OS/2 Version 1.2 and have long names (including device names) filtered according to the following rules:

- Any name not representable in the 8.3 file name format is not returned from DosFindFirst or DosFindNext. This is because the application's buffers are most likely not large enough to handle longer names.
- Any long file name passed to the file system API listed below is rejected in exactly the same way as under previous versions of OS/2. It is not acceptable to create and manipulate a name that you cannot find.

 DosOpen
 DosDelete
 DosMove
 DosQPathInfo
 DosSetPathInfo
 DosQFileMode
 DosSetFileMode
 DosMkDir
 DosRmDir
 DosFindFirst
 DosFindNext
 DosQFSAttach
 DosFSAttach
 DosCopy
 DosSearchPath.
- Long file names may be passed to DosChDir and DosQCurDir so that all programs can use all directories.
- Long names used with non-file system API (for example, DosCreateSem) are not filtered.

Moving Files with Long Names

Files that have long names cannot be moved to media managed by a non-installable file system without having their names modified, or truncated. The OS/2 File Manager, however, provides a facility for copying IFS files to non-IFS media and for returning these files to the IFS media with the long name intact.

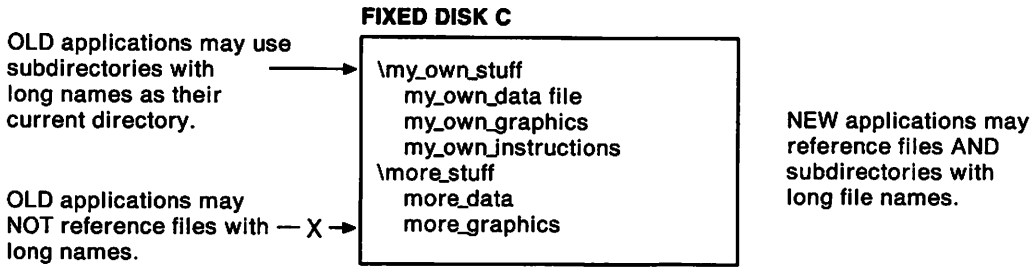


Figure 33-4. Accessing file objects with long names

Handling Long File Names

Because long file names may be input to applications through program command lines, dialog boxes, or API, applications must provide their users with rules for how to enter file names.

Users often supply file names as part of a program's command line or in response to a prompt from the program. Traditionally, users have been able to supply more than one file name on the command line or in a prompt by separating the names with certain characters, such as a blank. For some file systems, however, traditional separators can be used as valid characters in a file name. This means that some additional conventions are required to ensure that a program processes all characters in a name.

When a program processes arguments (including file names) from its command line, the program should treat the double quotation mark (") and the caret (¬) as quotation characters. For the double quotation mark, all characters between the starting and closing marks should be processed as a single argument. For the caret, the character immediately following the caret should be processed as part of the current argument. In both cases, the quotation characters are discarded and not treated as part of the final argument.

When a program processes two or more file names from a dialog box or other prompt, the program should expect the user to enter each file name on a new line. For example, a Presentation Manager application can use a multiple-line entry-field control to prompt for multiple file names. This makes the use of quotation characters unnecessary.

When a program is started from the File Manager, the File Manager may construct a command line for the program. If the command line includes file names, the File Manager separates each argument with a space and marks the end of the argument list with two null characters. Programs that start other programs using the DosExecPgm function can also pass argument using this convention or by using the quotation characters. In practice, most programs expect the command line as a single null-terminated string. Therefore, programs using DosExecPgm should prepare command lines that consist of a single string in which file names are quoted.

Processing Meta Characters

Meta characters are special characters used to reference a set of file objects with a common base name. For example,

```
*.EXE
```

can be used to reference the set of all files with the extension EXE. Meta characters include the asterisk (*) and question mark (?), and are also referred to as global file name characters. Meta characters work as follows for all programs, regardless of level number. Meta characters are illegal in all but the last element of a full path name.

Meta characters have two sets of semantics:

Search characters
: which are used to select which file names are returned to the user.

Edit characters
: which are used to construct a new name given a source name and a target name specification.

Edit characters are used, for example, in global copy commands.

```
copy *period.txt  *.old
```

Therefore, both "*" and "?" have two rules, one for searching and one for copy-editing (for example, in "ren <name-with-metas> <name-with-metas>"). "." has no special meaning itself in searching, but ? gives it one. "." has a special meaning for editing.

Searching for File Object Names

"*" matches 0 or more characters, any character, including blank. It does not cross NULL or \, which means it only matches a file name, not an entire path.

"?" matches 1 character, unless what it would match is a "." or the terminating NULL, in which case it matches 0 characters. It also does not cross \.

Any character other than "*" and "?" matches itself, including '.'.

Searching is case-insensitive. For example, "file.txt" references the same file named "file.TXT".

For compatibility reasons, any file name that does not have a dot in it gets an implicit one automatically appended to the end during searching operations. This means that searching for "file." would return "file".

Some file system function calls allow file object name specifications using meta characters. Refer to the *Control Program Programming Reference* for details on the use of meta characters with file system function calls.

Editing File Object Names

Meta characters in a source name simply match files and behave just like any other search meta character (see above).

Meta characters in a target name are copy-edit commands and work as follows:

- "?" copies one character unless what it would copy is a "."; in which case, it copies 0. It also copies 0 characters if we are at the end of the source string.
- "*" copies characters from the source name to the target name until it finds a source character that matches the character following it in the target.
- "." in the target name causes the source pointer to match the corresponding "." in the target. They count from the left.

Editing is case-insensitive. If a case conflict between the source and editing string arises, the case in the editing string is used:

```
copy file.txt *E.tmp
```

results in file.txt being copied as filE.tmp.

The DosEditName function call provides applications with the ability to transform a file object name into another name, using an editing string that contains global characters. See "Transforming a Name" on page 33-11.

Naming Character Devices

Naming conventions for character devices are similar to those for naming files. OS/2. has reserved certain names for character devices supported by the base device drivers. These device names are listed below:

COM1 – COM3	First through third serial ports
CLOCK$	Clock
CON	Console keyboard and screen
SCREEN$	Screen
KBD$	Keyboard
LPT1	First parallel printer
LPT2	Second parallel printer
LPT3	Third parallel printer
NUL	Nonexistent (dummy) device
POINTER$	Pointer draw device (mouse screen support)
MOUSE$	Mouse.

These names can be used in the DosOpen function call to open the corresponding devices. Reserved device names take precedence over file names; DosOpen checks for a device name before checking for a file name. However, COM1 through COM3 are reserved device names only when the ASYNC (RS-232C) Device Driver is loaded. The same is true for POINTER$ and MOUSE$, which are reserved only when a mouse device driver is loaded. Do not use a file name which is the same as a reserved device name. The file never will be opened, because the command opens the device instead.

An application should call DosQFHandState to verify that a file or device has been opened. See "Determining the State of an Open File or Device" on page 33-14.

Managing Names of File Objects

The following function calls allow applications to manage names of OS/2 file objects:

DosQSysInfo
: Returns the maximum path length supported by the system

DosEditName
: Transforms a file name.

Querying the Maximum Path Length

An application that recognizes long file names may be run on systems with or without installable file systems. In addition, the maximum length of a file name may vary from one installable file system to another. So that an application can properly handle this variable condition (for example, allocating large enough buffers to hold file names), the application should call **DosQSysInfo** to determine the maximum path length supported by the system.

Note: This call should be made before calling file system functions that require full path names.

Transforming a Name

File system API that allow an application to copy, rename or move file objects do not support the use of global characters. Whereas, for example, a user may do a global copy of all files with the extension .EXE by entering

```
COPY *.EXE
```

on the OS/2 command line, an application cannot do a similar global copy operation by making a single function call (DosCopy or DosMove). These function calls operate on a single, specific file object.

DosEditName, however, provides applications with the ability to transform an element of a full path name into another name, using an editing string that contains global characters. For example, an application that wants to copy all files with an extension of .SRC as files with an extension .SAM may first search for all files with that extension (using DosFindFirst/ DosFindNext), transform the file names (using DosEditName with an editing string of *.SAM) and then copy the files with the new extension (using DosCopy).

Creating File Objects

The following function calls allow applications to create OS/2 file objects (files and subdirectories):

DosOpen
: Opens an existing, or creates a new file

DosOpen2
: Opens an existing, or creates a new file with extended attributes

DosMkDir
: Creates a new subdirectory

DosMkDir2
: Creates a new subdirectory with extended attributes.

Creating a File

DosOpen is used to establish a connection between an object and an application. This connection is in the form of a 16-bit identifier called a file handle that is used to reference the object and any information associated with the object.

DosOpen returns a handle that is used to access the object in other file system calls. The object can be a new file, an existing file, or a replacement for an existing file. The application uses the OpenFlag parameter to indicate what action is to be taken when the object name specified already exists, or does not exist. (For example, if the file already exists, the application may wish to replace the file.)

The object can also be a character device, a block device, or the client end of a named pipe. The FileName parameter of DosOpen identifies the object by means of a file name for a file or a named pipe, a character device name, or a drive specification.

Note: If the object is a named pipe, the pipe must be in a listening state for the DosOpen to be successful. For more information on opening a named pipe, see "Gaining Access to the Named Pipe" on page 35-13.

Sample Programs: Examples of the **DosOpen** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\IMAGE\IMAGE.C
```

Specifying a File Size

If the object to be opened is a new file or a replacement for an existing one, the FileSize parameter is used to specify the file size in bytes. The file system attempts to allocate contiguous space on the media for the new file. If the file being opened already exists, this parameter is ignored. After a file has been opened, its size can be changed by calling DosNewSize. See "Changing the Size of a File" on page 33-13.

Setting File Attributes

The FileAttribute parameter of DosOpen determines what attributes a file has if the file is being created or replaced. These include:

- Read-only file
- Hidden file
- System file
- Subdirectory
- File Archive.

Attributes of a closed file can be queried by calling DosQFileMode and can be modified by calling DosSetFileMode. See "Manipulating File Attributes" on page 33-13.

Creating a File with Extended Attributes

Extended attributes may be defined by an application when a file object is created. An application may define an extended attribute for a file when the file is created during a **DosOpen2** call. See "Defining Information Associated with File Objects" on page 33-16.

Setting File Handle State Flags

The OpenMode parameter of DosOpen sets flags indicating characteristics of the file handle. The flags that can be set include:

Flag	Purpose
Inheritance	Handle is inherited by a child process created with DosExecPgm, or is private to the calling process.
Write-through	Actual I/O for synchronous writes is not guaranteed as complete or is guaranteed as complete before the write returns.
Fail-Errors	Media errors are reported by the system critical error handler, or by the application.
DASD	FileName parameter is the name of a file or device opened in the normal way, or a drive specification for a fixed disk or diskette drive.
Cache	The disk device driver caches or does not cache data for I/O operations on the file.

The DASD flag can be set for direct access to an entire disk or diskette volume, independent of the file system. When the DASD flag is set, the handle returned by DosOpen represents the logical volume as a single file. To block other processes from accessing the logical volume, a DosDevIOCtl Category 8, Function 0 call should be made using the handle returned by DosOpen. The DASD flag should be set only by systems programs, not applications.

After an object has been opened, its file handle state flags may be queried by calling DosQFHandState and reset by calling DosSetFHandState. See "Determining the State of an Open File or Device" on page 33-14.

Controlling Access to a File

The OpenMode parameter of DosOpen also includes flags to indicate sharing and access rights to the file for processes that are sharing files.

An application defines its file access rights (that is, I/O it needs to perform on a file) by setting the appropriate file access mode field in the OpenMode parameter. An application accesses a file as:

- Read-Only
- Write-Only
- Read/Write.

An application defines what I/O operations other processes may perform on a file by setting the appropriate sharing mode field in the OpenMode parameter. Other processes are granted:

- Deny Read/Write access
- Deny Write access
- Deny Read access
- Deny Neither Read or Write access (Deny None).

When a child process inherits a file handle, it also inherits the file's sharing and access modes.

File sharing is based on cooperation between the sharing processes. For example, if process A calls

DosOpen to open a file with Deny Write sharing and process B calls DosOpen to open the same file with Read/Write access, the DosOpen call made by process B fails.

Note: For named pipes, the access and sharing modes must be consistent with those specified by DosMakeNmPipe.

Applications may also control access to specific regions within a file by calling DosFileLocks and DosFileIO. See "Controlling Access to a Region within a File" on page 33-32.

Creating a Subdirectory

To create a new subdirectory, an application calls **DosMkDir** and specifies a directory path name. If the call is successful, a new subdirectory is created at the end of the path on the specified disk. If no path is specified, a new subdirectory is created at the end of the current directory for the process. If any subdirectories in the path do not exist, the subdirectory is not created.

Because a subdirectory is a file object, an application may define an extended attribute for the directory when it is created during a **DosMkDir2** call. See "Defining Information Associated with File Objects" on page 33-16.

Managing File Objects

File system function calls allow applications to manage OS/2 file objects:

- Files
- Directories.

Managing Files

The following function calls allow applications to manage OS/2 files:

DosSetFileMode
Sets file attributes
DosQFileMode
Returns file attributes
DosNewSize
Changes the size of a file
DosMove
Moves a file from one subdirectory to another
DosCopy
Copies a file or subdirectory
DosDelete
Deletes a file.

Manipulating File Attributes

DosSetFileMode can be used to change some of a file's attributes. Those that can be changed are indicators for:

A read-only file
A hidden file
A system file
An archived file.

File attributes that cannot be changed are indicators for:

The volume label
A subdirectory.

DosQFileMode may be used to query the current setting of file attributes.

Changing the Size of a File

DosNewSize allows an application to extend or truncate the size of a file that is open for Read/Write or Write-Only access. To change the size of a Read-Only file, an application must first call DosSetFileMode, to set the Read-Only file attribute to zero.

When a file is extended, the value of the additional bytes allocated by the system is undefined.

Moving a File to Another Subdirectory

DosMove is used to move a file from one subdirectory to another on the same drive. In the process of moving the file, its name can be changed. Global file name characters are not permitted.

Copying a File or Subdirectory

When copying a file, specify the **DosCopy** function when the source file and target file are on different drives. However, when a subdirectory is specified to be copied and an I/O error occurs, the specified file being copied from the source file to the target file at the time of the error will be deleted from the target path and DosCopy will be terminated. Also, when a non-directory file is specified to be copied, and an I/O error occurs, the file will be deleted from the target path in replace mode, re-sized to its original size in append mode, and the function terminated.

DosCopy will copy the source file's attributes, such as date of creation, and time of creation to the target file. Additionally, DosCopy will copy the source file's extended attributes when creating a new subdirectory, when creating a new file, or when replacing an existing file.

Deleting a File

Issuing **DosDelete** removes the directory entry associated with a file name. Global file name characters are not permitted.

Files whose read-only attribute is set cannot be deleted. To change the setting of the read-only bit, use DosSetFileMode.

Managing File Handles

The following function calls allow applications to manage OS/2 file handles:

DosDupHandle
Creates a duplicate file handle
DosQFHandState
Returns the state of an open file or device
DosSetFHandState
Sets the state of an open file or device
DosQHandType
Returns the handle type
DosSetMaxFH
Sets the maximum number of handles
DosClose
Closes a handle.

Creating a Duplicate Handle

DosDupHandle enables a process to create a duplicate handle for an open file, pipe, or device.

The value for the OldFileHandle parameter is the handle of an open file, pipe, or device. The valid values for NewFileHandle include FFFFH, 0000H (standard input), 0001H (standard output), and 0002H (standard error). Any value other than FFFFH is assumed to be the value of the new file handle.

A value of FFFFH causes the system to allocate a new file handle and send it to this location. If the value specified for NewFileHandle is that of a currently open file, the file handle is closed before it is redefined.

An agreed upon value for a duplicate file handle can be established between a parent process and a child process. Avoid choosing arbitrary values for NewFileHandle.

The duplicate handle acquires the characteristics of the original. If you move the read/write pointer of the original file handle, for example by issuing DosRead, DosWrite, or DosChgFilePtr, the pointer of the duplicate handle is also moved. If the original handle has access to regions in a file that have been locked by DosFileLocks, the duplicate also has access.

If inheritance was indicated when a file was opened with DosOpen, a parent process can create a duplicate handle for the file and pass it to a child process by means of shared memory. This permits the child to close the duplicate handle without affecting the file handle of the parent.

Because a parent process controls the meanings for standard I/O done by any child process it creates, the parent can use DosDupHandle to redefine unnamed pipe handles as standard I/O handles to communicate with a child.

The parent process creates two pipes and duplicates the read handle of one pipe as 0000 and the write handle of the other pipe as 0001. When the child process performs standard I/O, instead of reading from the keyboard and writing to the display, it reads from and writes to the pipes created by its parent. As the owner of the pipe, the parent process uses its read and write handles to write to the pipe defined to the child as standard input and read from the pipe defined to the child as standard output.

For information on creating an unnamed pipe, see "Passing Data in an Unnamed Pipe" on page 35-7.

Determining the State of an Open File or Device

After a file has been opened, the file handle state flags set with a DosOpen can be queried and reset by issuing **DosQFHandState** and **DosSetFHandState**. For the values of these flags, see "Setting File Handle State Flags" on page 33-12. The handle returned by DosSetFHandState is used for subsequent input and output to the file.

Here are two examples, using this call.

Example 1: Suppose an application needs to have data written in a specific order. To guarantee the order of the data written, it must perform separate synchronous write operations. The application can issue DosSetFHandState to set the synchronous I/O flag. This action does not affect any previous asynchronous writes, whose data can remain in the buffers.

Example 2: Suppose an application cannot handle a critical error situation. DosSetFHandState can be issued to reset critical error handling as the system's responsibility. The I/O function call that caused the critical error must be reissued so the error can recur, causing control to be passed to the system. In the case where asynchronous I/O is being done, the exact time when the result of this call is available to the application is unpredictable.

Determining a Handle Type

DosQHandType allows an application to determine whether a handle is to a file, pipe, or device. This call can be used when a file-oriented application needs to modify its behavior, depending on the source of its input. For example, CMD.EXE suppresses writing prompts when its input is from a disk file.

Setting the Maximum Number of Handles for a Process

The system initially allots 20 file handles to a process, which is the recommended maximum amount for an application. However, if the system limit has not been reached, this amount can be increased by issuing **DosSetMaxFH**. When this call is made, all currently open file handles are preserved.

Closing a Handle to a File, Pipe, or Device

DosClose closes a handle to a file, pipe or device. If one or more additional handles to a file have been created with DosDupHandle, the internal buffers for the file are not written to disk and its directory entry is not updated until DosClose has been issued for all the handles.

For information on the effects of a DosClose issued with a named pipe handle, see "Ending Access to the Named Pipe" on page 35-14.

Sample Programs: Examples of the **DosClose** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\IMAGE\IMAGE.C
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Managing Directories

The following function calls allow applications to manage OS/2 directories:

DosQCurDir
: Returns the current directory

DosChDir
: Changes the current directory

DosRmDir
: Deletes a directory.

Determining the Current Directory

Issuing **DosQCurDir** returns the full path name of the current directory for the specified drive. The string does not begin with a back slash (\) and it ends with a byte containing 00H.

Sample Programs: Examples of the **DosQCurDir** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\C\SAMPLES\PM\IMAGE\IMAGE.C
\TOOLKT12\MASM\SAMPLES\BSE\CRERRB\CRERR.ASM
```

Changing the Current Directory

To change the current directory, issue **DosChDir** with the path name of the directory you want as the current one.

Removing a Subdirectory

A subdirectory cannot be removed if it is the current subdirectory, or if it contains hidden files or directory entries other than the "." and ".." entries. When these requirements for removal are met, **DosRmDir** can be issued with a path name to remove the subdirectory from the disk.

Sample Programs: Examples of the **DosChDir** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
```

Managing Logical Drives

The following function calls allow applications to manage OS/2 logical drives:

DosQCurDisk
Returns the default drive
DosSelectDisk
Sets the default drive
DosSetFSInfo
Set drive information
DosQFSInfo
Return drive information
DosSetVerify
Set write verification
DosQVerify
Determine write verification.

Determining the Default Drive

An application may query the default drive by calling **DosQCurDisk** An application may specify the default drive by calling **DosSelectDisk.**

Determining Drive Information

An application may specify or query the free space on a drive, as well as the volume label of a drive. To specify the drive information, an application calls **DosSetFSInfo.** To query the drive information, an application calls **DosQFSInfo.**

Determining Write Verification to a Drive

DosSetVerify allows you to set a data verification switch. When the switch is set, each time a write is made to the disk, the data is verified to ensure that it has been recorded correctly. Recording errors are rare; this function is provided for recording critical data.

DosQVerify determines the current state of the switch.

Defining Information Associated with File Objects

OS/2 file systems maintain a standard set of information on file objects. This information includes:

```
Date of creation
Time of creation
Date of last access
Time of last access
Date of last write
Time of last write
```

as well as the name of the file object. A subset of this information is typically displayed by entering the DIR command on the OS/2 command line. This standard set of information is referred to as Level 1 File Information. Applications may obtain Level 1 File Information by calling DosQPathInfo and DosQFileInfo. Applications may set Level 1 File Information by calling DosSetPathInfo and DosSetFileInfo. See "Managing Information Associated with File Objects" on page 33-23.

Applications may also attach additional information to a file object in the form of an extended attribute (EA). Extended attributes can be used to describe the file object to:

- Another application
- The operating system
- The file system managing that object.

This information can be used to:

- Store notes on file objects (for example, the name of the file creator)
- Categorize file objects (for example, source, samples, icons, bit maps)
- Describe the format of data contained in the file object (for example, a data record)
- Append additional data to the file object.

A file object may have more than one extended attribute associated with it. See Figure 33-5 on page 33-17. EAs associated with a file object are not part of a file object's data, but are maintained separately and managed by the file system that manages that object.

Note: Extended attributes are supported by the OS/2 Version 1.2 FAT file system.

Each extended attribute consists of:

- A name
- A value.

The maximum size of an EA is 64KB.

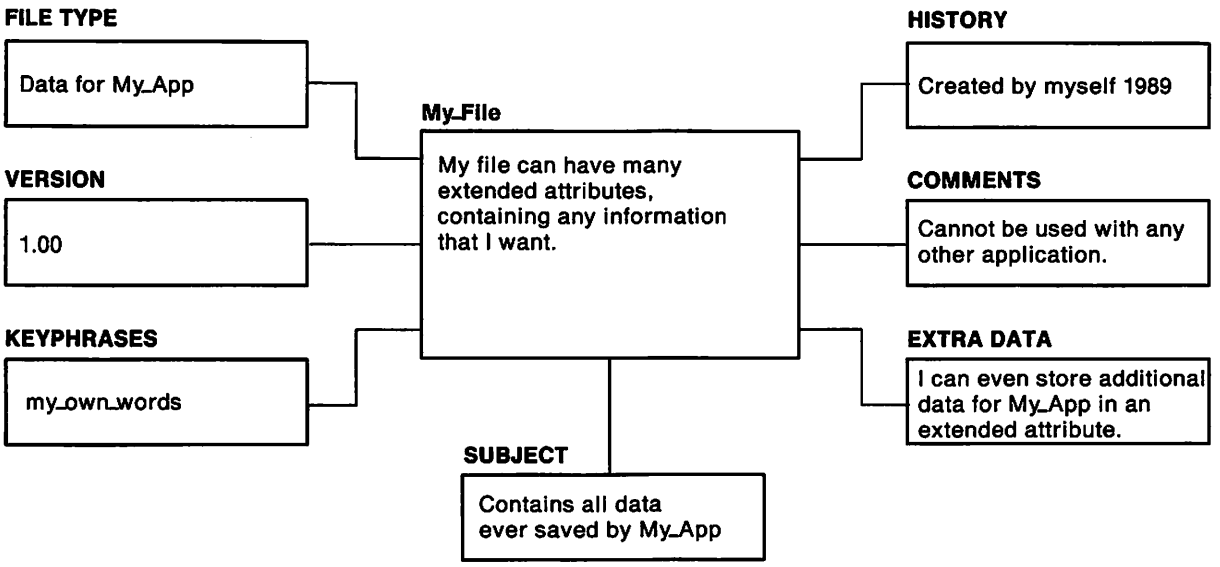


Figure 33-5. A file's extended attributes

An EA name consists of ASCII text, chosen by the application developer, that is used to identify a particular EA. EA names are restricted to the same character set as a file name.

An EA value consists of arbitrary data, that is, data of any form. Because of this, OS/2 does not check data that is associated with an EA.

So that EA data is understandable to other applications, conventions have been established for:

- Naming EAs
- Indicating the type of data contained in EAs.

In addition, a set of standard EAs (SEAs) have been defined. SEAs define a common set of information that can be associated with most files (for example, file type and file purpose). Through SEAs, many applications can access the same, useful information associated with files. For example, users may use an option of the OS/2 File Manager to browse SEAs associated with files.

Note: Applications are not limited to using SEAs to associate information with files. They may define their own application-specific extended attributes.

Applications define and associate extended attributes with a file object through file system function calls. See "Managing Extended Attributes" on page 33-23 for more information.

Naming EAs

The name of an application-specific EA is prefixed with the name of the company (or suitable abbreviation) and the name of the product (or suitable abbreviation) that defines the attribute. For example, Company A has an OS/2 Application B that defines extended attributes STUFF, MORE_STUFF, and STILL_MORE_STUFF for its file objects. The names of these extended attributes may be represented by:

```
AB.STUFF  AB.MORE_STUFF  AB.STILL_MORE_STUFF
```

The name of a standard EA has a dot (.) as a prefix. This identifies the EA as a standard EA. No application should define EAs that start with a dot, as additional SEAs may be added at a later time. The prefixes $ @ & and + are reserved for system usage.

The names of standard EAs that have been defined for OS/2 Version 1.2 are:

.TYPE
.KEYPHRASES
.SUBJECT
.COMMENTS
.HISTORY
.VERSION
.ICON
.ASSOCTABLE
.CODEPAGE.

See "Using Standard Extended Attributes" on page 33-19 for a detailed description of each standard extended attribute.

Indicating EA Value Data Types

The value of an extended attribute may contain any type of data. So that applications may understand the type of information stored in an EA, the first WORD of EA data should contain one of the data types found in Table 33-1 on page 33-19.

When a type is length-preceded, a word precedes the data indicating the length, in bytes. For example, the string 'Hello' is represented in the following way:

```
EAT_ASCII 0005 Hello
```

Note: All user-defined data types should be length-preceded.

Using Multi-valued Data Types

An extended attribute may contain more than one piece of information. For example, an EA may store a list of names that a piece of mail was sent to. Multi-valued data types are used to specify how these individual pieces of data are grouped.

When an EA is multi-valued, the first entry in the list is assumed to be the default, when the concept of a default applies. For example, suppose the value of a .TYPE EA contains the ASCII strings 'Text' and 'C Code'. 'Text' is the default type. If 'C Code' were the first in the list ('C Code' and 'Text') then 'Code' would be the default.

Because data entries are length-preceded, it is very easy to traverse a multi-valued list.

In order to allow EAs of different codepages, multi-valued EAs include a field in which the EA's codepage is specified. For example, the codepage field could be used to indicate that the comments on a Kanji file are written in Spanish. If this value is 0, the file default is assumed. For more information on codepage and EAs, see ".CODEPAGE EAs" on page 33-22.

Note: Codepage data is for use by applications. OS/2 does not examine or use EA codepage information.

There are three multi-valued EA data types. Of these, only the first can be used in standard extended attributes. Application-specific EAs can use any of these data types.

Multi-valued, Multi-typed Data: This data type indicates that the value of a single EA contains several pieces of information, each of a different data type. It is formatted as follows:

```
EAT_MVMT  Codepage  NumEntries  [data_type data ] ...
WORD      WORD      WORD         WORD
```

where the first word indicates that the EA value is multi-valued, multi-typed. The second word indicates the codepage associated with the language in which the EA value is written. The third word indicates the number of entries contained in this EA value. The next word indicates the data type for the first entry in this EA value, followed by the data for the first entry. The next word, if any, indicates the data type for the second entry in this EA value, followed by the data for the second entry. And so forth.

Whether or not the data is length-preceded is a function of the data type. For example, an extended attribute may have the following value:

```
EAT_MVMT  0000  0002  EAT_ASCII   000A Hello John
                      EAT_BINARY  0003 12 21 34
```

This is a multi-valued extended attribute with two entries, using the default codepage. The first entry is the string 'Hello John', and the second is the binary data 12 21 34.

Multi-valued, Single-type Data: This data type indicates that the value of a single EA contains several pieces of information, each of the same data type. For example:

```
EAT_MVST  Codepage  NumEntries  Data_type  [data ...
WORD      WORD      WORD        WORD
```

where the first word indicates that the EA value is multi-valued, single-typed. The second word indicates the codepage associated with the language in which the EA value is written. The third word indicates the number of entries contained in this EA value. The next word indicates the data type of all the entries contained in this EA value, followed by the data for all entries.

Whether or not the data is length-preceded is a function of the data type. For example, the following EA value contains three ASCII names:

```
EAT_MVST 0000 0003 EAT_ASCII 0004 Mark
                             0005 Ellen
                             0003 Liz
```

where each name string is preceded by the length of the string.

Table 33-1. EA data types. Constants are defined in the BSEDOS.H and BSEDOS.INC files that are included on the diskettes accompanying the OS/2 Programming Tools and Information.

Constant	Data Type	Description
EAT_BINARY	FFFE	length-preceded binary
EAT_ASCII	FFFD	length-preceded ASCII
EAT_BITMAP	FFFB	length-preceded bitmap
EAT_METAFILE	FFFA	length-preceded metafile
EAT_ICON	FFF9	length-preceded icon
EAT_EA	FFEE	ASCII EA name of associated data
EAT_MVMT	FFDF	multi-valued, multi-typed data
EAT_MVST	FFDE	multi-valued, single-typed data
EAT_ASN1	FFDD	ASN.1 field
	8000-	Reserved
	0000-7FFF	User definable

ASN.1: This data type indicates that the extended attribute uses the ASN.1 ISO standard to describe a multi-valued data stream.

Using 'Include' Data Types

EA values may contain pointers to data stored in other places.

EAT_EA: This data type indicates that the data contained in this EA value is continued (included) in another EA associated with the file object. Among other things, this allows EAs larger than 64KB and up to the total EA size to be constructed. For example, the following EA value contains the string 'Hello', followed by the data in the EA named AB.STUFF, followed by the string 'Bye'

```
EA_MVMT 0000 0003 EAT_ASCII 0005  Hello
                  EAT_EA    0008  AB.STUFF
                  EAT_ASCII 0003  Bye
```

Using Standard Extended Attributes

The following section describes the standard extended attributes (SEAs). The values of standard extended attributes can be multi- or single-valued, with formats following the data type conventions already discussed.

Note: Where entries for standard EAs consist of ASCII characters, case is important.

.TYPE EAs: This extended attribute indicates the file object's type and is used similar to DOS file extensions. The name of this EA consists of the string '.TYPE'. The value of this EA contains the file object's type.The following file object types are predefined:

Plain Text
OS/2 Command File
DOS Command File
Executable
Metafile
Bitmap
Icon
Binary Data
Dynamic Link Library
C Code
Pascal Code
BASIC Code
COBOL Code
FORTRAN Code
Assembler Code
Library
Resource File
Object Code.

Data files only need to identify file type. For data files without EAs, the file type is derived from the file extension (if there is one).

File object types are represented as length-preceded ASCII strings, uniquely identifying the file object type. This identifier is referenced within the program file .ASSOCTABLE EA in order to bind the data file type to the program file. It is important that this name be a unique identifier because all file type names are public data. For example, if application A and application B both had a type name of SPREADSHEET, the filing system would not be able to identify A's SPREADSHEET from B's SPREADSHEET.

Applications can define their own file object types. The recommended convention for defining file object types is:

```
Company_name
Application_name
Application-specific_name
```

with spaces separating each. For example, spreadsheet files generated by My_Application written by My_Company may have a file object type of

```
My_Company My_Application spreadsheet
```

.KEYPHRASES EAs: This extended attribute contains text keyphrases for a file object. The name of this EA consists of the string '.KEYPHRASES'. The value of this EA contains the text keyphrases. Keyphrases can be used for a database-type search or to help the user understand the nature of the file.

Keyphrases are represented as ASCII characters. Multiple keyphrases can be stored in the value of this extended attribute, each stored in a separate entry in a multi-valued field. For example, the following extended attribute contains three keyphrases:

```
EAT_MVST 0000 0003 EAT_ASCII 0008 ABC Inc.
                   EAT_ASCII 000A Salesman A
                   EAT_ASCII 000F Product X sales
```

.SUBJECT EAs: This extended attribute contains a brief summary of the content or purpose of a file object. The name of this EA consists of the string '.SUBJECT'. The value of this EA contains the purpose of the file object and consists of a single-valued ASCII string, less than 40 characters in length.

.COMMENTS EAs: This extended attribute contains miscellaneous notes about a file object (for example, peculiarities, restrictions, requirements). The name of this EA consists of the string '.COMMENT'. The value of this EA contains miscellaneous notes and can be multi-valued and of any type.

.HISTORY EAs: This extended attribute contains the modification history for a file object, indicating the author of the file and all subsequent changes. The name of this EA consists of the string '.HISTORY'. The value of this EA contains the modification history for a file object and can be multi-valued, with each action entry described in a separate field. Each entry consists of ASCII characters with the following format:

```
PERSON  ACTION(created, changed or printed)  DATE
```

For example, the following .HISTORY extended attribute contains two action entries:

```
EAT_MVMT 0000 0002
         EAT_ASCII 0017 Joe    Created  2/10/88
         EAT_ASCII 0017 Harry  Changed  2/11/88
```

Because the size of this extended attribute is potentially large, an application may let the user decide when an entry is added to this extended attribute, to avoid unwanted growth. For example, there are some cases when it is important to note when a document is printed. However, it is probably unnecessary to note every time the file is printed.

.VERSION EAs: This extended attribute contains the version number of the file object format, for example:

```
My_Application 1.0
```

The name of this EA consists of the string '.VERSION'. The value of this EA contains the file object version number and consists of ASCII characters. This extended attribute should only be modified by the application that creates the file object.

.ICON EAs: This extended attribute contains the physical icon data used to represent the file object in the File Manager. The name of this EA consists of the string '.ICON'. The value of this EA contains the physical icon data and has the following format:

```
EAT_ICON  data_length  data
WORD      DWORD
```

The data is of type BITMAPARRAYFILEHEADER and is used to specify an array of device-dependent and one device-independent icon/pointer/bit map. The GpiLoadBitmap and WinLoadPointer function calls support this icon file format.

The Icon Editor is used to create the icon, which is saved in a icon file. The .ICON extended attribute for a program file is created by the Resource Compiler (RC) as part of the compile process of making a program file by specifying the DEFAULTICON keyword.

```
DEFAULTICON <file name.ico>
```

This keyword uses the icon definition contained in the specified icon file (file name.ico) to create the .ICON EA for the program file.

Program files should store the binary icon data in an .ICON EA. They should use the .ASSOCTABLE EA to install icons for data files.

.ASSOCTABLE EAs: This extended attribute is used to provide information on a program file. The name of this EA consists of the string '.ASSOCTABLE'. The value of this EA contains program file information and consists of multi-valued, multi-typed fields.

The value of a program's .ASSOCTABLE EA contains a table that links the program file with the file type (that is, the value of a .TYPE EA), file extension, and icon data for data files that it generates or references. Because programs may understand and reference data files generated by other programs, this table can be used to link a program with those files.

The .ASSOCTABLE extended attribute tells the OS/2 File Manager:

- What icons to display for data files with a particular .TYPE EA
- What icons to display for data files that do not have a particular .TYPE EA but do have a particular file extension.

In addition, the .ASSOCTABLE EA tells the File Manager which data file the user may select for execution to automatically run the program file.

The value of this extended attribute is multi-valued and multi-typed, with an entry describing each associated file. Each entry in the value of this extended attribute is also multi-valued and multi-typed and has the following format:

```
EAT_MVMT 0000 0004 EAT_ASCII   .TYPE name
                   EAT_ASCII   file extension
                   EAT_BINARY  flags
                   EAT_ICON    icon data
```

where

for example, My_Company's My_Application generates or references data files that have the .TYPE names:

```
My_Company My_Application documentation
My_Company My_Application macros
My_Company My_Application spreadsheet
My_Company My_Application chart
Your_Company Your_Application forecast
```

Figure 33-6 on page 33-22 illustrates what the value of the .ASSOCTABLE EA for My_Application might look like. It is a multi-valued, multi-typed EA with five multi-valued, multi-typed entries (one for each file type referenced or generated by the application).

An .ASSOCTABLE extended attribute for a program file is created by the Resource Compiler (RC) as part of the compile process of making a program file. The source for the .ASSOCTABLE EA is contained in the resource file for an application:

```
ASSOCTABLE
BEGIN
'association name','extension',flags,icon file name
'association name','extension',flags,icon file name
:
END
```

Association name is a name of the file type this program understands. (This is the same name found in the .TYPE field of data files.) Extension is the three-letter file extension that is used to identify files of this type, if they have no .TYPE EA entry. (This may be empty.) Icon file name is the name of the file that contains the icon used to represent this file type. (This may also be empty.)

.ASSOCTABLE flags are defined as follows:

- EAF_DEFAULTOWNER

 This flag indicates that the program file is the default application for data files with the specified type. This determines what program is run by the File Manager when the file is double clicked on.

 If more than one program has marked that it is the EAF_DEFAULTOWNER for a particular data file .TYPE, the File Manager will not know what program to run when the file of this .TYPE is double clicked on. Also, if no program has marked that it is the EAF_DEFAULTOWNER for a particular data file .TYPE, the File Manager will be similarly confused. In both cases, the File Manager provides the user with a list of program files that understand the file .TYPE, regardless of whether the program file is the owner or not. The user selects the program to run from this list.

- EAF_UNCHANGEABLE

 If this flag is set, the entry describing data files of this type is not editable.

```
EAT_MVMT   0000 0005                                    ; there are 5 associated file types

EAT_MVMT   0000 0004                                    ; description of 1st associated file type
EAT_ASCII  0027 My_Company My_Application documentation ; file type
EAT_ASCII  0003 DOC                                     ; file extension
EAT_BINARY flags                                        ; flags
EAT_ASCII  icon data                                    ; physical icon data

EAT_MVMT   0000 0004                                    ; description of 2nd associated file type
EAT_ASCII  0020 My_Company My_Application macros
EAT_ASCII  0003 MAC
EAT_BINARY flags
EAT_ICON   icon data

EAT_MVMT   0000 0004                                   ; description of 3rd associated file type
EAT_ASCII  0025 My_Company My_Application spreadsheet
EAT_ASCII  0003 SPR
EAT_BINARY flags
EAT_ICON   icon data

EAT_MVMT   0000 0004                                   ; description of 4th associated file type
EAT_ASCII  001F My_Company My_Application chart
EAT_ASCII  0003 CHT
EAT_BINARY flags
EAT_ICON   icon data

EAT_MVMT   0000 0004                                   ; description of 5th associated file type
EAT_ASCII  001F Your_Company Your_Application forecast
EAT_ASCII  0003 FOR
EAT_BINARY flags
EAT_ICON   icon data
```

Figure 33-6. Value of .ASSOCTABLE EA for My_Application

- EAF_REUSEICON

 This flag is specified if a previously defined icon in the ASSOCTABLE is to be reused. Entries with this flag set have no icon data defined. The icon used for this entry will be the icon used for the previous entry.

EAF_ flags are defined in the PMWIN.INC and PMWIN.H files that are included on the diskettes accompanying the OS/2 Programming Tools and Information. EAF_ flags may be OR'ed together when specified in the ASSOCTABLE.

The source for the .ASSOCTABLE extended attribute in the resource file for My_Application may look like Figure 33-7 on page 33-23. My_Application can load and use some files generated by Your_Application. However, because My_Application is not the default owner of those files, the File Manager does not run My_Application when those files are double clicked on.

.CODEPAGE EAs: This extended attribute contains the codepage for the file. If this extended attribute is not provided, the codepage of the file is the system default or defined by the application.

The codepage of the EA data associated with the file is assumed to be that of the file, unless the EA entry is specifically overridden in the codepage field in the multi-valued extended attribute data type.

Relating Programs to Icons to Type

The .TYPE extended attribute is very important. It indicates what type of data is in a file. It also implies what programs can edit the file, and what icon should be used for the file.

The .ASSOCTABLE extended attribute allows an application to indicate the type, extension, and icon for data files that it recognizes. It also contains an ownership flag. This data can be automatically installed by the File Manager.

```
ASSOCTABLE
BEGIN
'My_Company My_Application documentation', 'DOC', EAF_DEFAULTOWNER, My_Application.ICO
'My_Company My_Application macros', 'MAC', EAF_DEFAULTOWNER+EAF_REUSEICON
'My_Company My_Application spreadsheet', 'SPR', EAF_DEFAULTOWNER+EAF_REUSEICON
'My_Company My_Application chart', 'CHT', EAF_DEFAULTOWNER+EAF_REUSEICON
'Your_Company Your_Application forecast', 'FOR', 0
END
```

Figure 33-7. Source for .ASSOCTABLE EA for My_Application

The File Manager can use the .TYPE EA to determine a default application to run and a default icon for a file of a particular type. If there is a .ICON EA, it will be used instead of the icon associated with a particular data type.

When a program knows how to use files created by different programs, it may want to install .ASSOCTABLE EA entries for those other programs.

Managing Information Associated with File Objects

The following function calls allow applications to manage information associated with OS/2 file objects:

DosOpen2
: Creates a file with extended attributes

DosMkDir2
: Creates a subdirectory with extended attributes

DosSetFileInfo
: Sets file information for an open file

DosQFileInfo
: Gets file information for an open file

DosSetPathInfo
: Sets information for a file object

DosQPathInfo
: Gets information for a file object

DosEnumAttribute
: Enumerates extended attributes for a file object.

Managing Extended Attributes

For any file object, an application may:

- create/set extended attributes
- get extended attributes.

EAs may be defined by an application when a file object is created. An application may define an EA for a file when the file is created during a **DosOpen2** call. Similarly, an application may define an EA for a directory when the directory is created during a **DosMkDir2** call.

EAs may also be defined by an application for existing file objects. An application may define an EA for an existing file object:

- By referencing the file object by its handle and calling **DosSetFileInfo**
- By referencing the file object by its name and calling **DosSetPathInfo**.

An application can look at the EAs for a file object:

- By referencing the file object by its handle and calling **DosQFileInfo**
- By referencing the file object by its name and calling **DosQPathInfo**
- By referencing the file object by its handle or name and calling **DosEnumAttribute**.

In addition, an application can search for file objects that have specific EA names by calling **DosFindFirst2**.

Controlling Access to EAs

Like the file objects they are associated with, extended attributes have multiple readers and writers. This means that one process may be querying EAs for a file object, while another is setting EAs for the same file object.

Operations on EAs are atomic: any one query or set operation completes before any other query or set operation can be performed on the same object. This guarantees that the EAs will remain in a consistent state, but does not guarantee the order in which the operations are done.

So that unpredictable results do not occur during multiple simultaneous operations to extended attri-

butes, sharing protection is provided. EA manipulation is associated with the access permission to the related file object (file or directory).

Handle-based access permission is controlled by the sharing/access mode of the associated file object.

1. If the file object is open for READ, querying EAs (DosQFileInfo) is allowed.
2. If the file object is open for WRITE, setting EAs (DosSetFileInfo) is allowed.

Path-based access permission is controlled by adding the file object to the sharing set for the duration of the call.

1. For querying EAs (DosQPathInfo), requested access permission is READ/DENY-WRITE.
2. For setting EAs (DosSetPathInfo), requested access permission is WRITE/DENY-ALL.

Note: The set/query EA function calls fail if another process holds conflicting sharing rights to the file object.

No explicit EA sharing is done for DosEnumAttribute. Implicit sharing exists if the caller passes in the handle of an open file, since sharing access to the associated file is required to modify its EA. No sharing is done if the caller passes in the path name. This means that if some other process is editing the EAs, and changes them between two calls to DosEnumAttribute, inconsistent results may be returned (for example, the same values may be returned twice, some values may be missed, etc.). To prevent the modification of EAs for the handle case, the file should be open in deny-write mode before calling DosEnumAttribute. To prevent the modification of EAs for the pathname case, the file should be open in deny-write mode before calling DosEnumAttribute. For the directory name case, no sharing is possible.

Defining EA data structures

There are a series of data structures through which OS/2 function calls manipulate extended attributes for applications:

- full EAs (FEAs)
- a list of full EAs (FEAList)
- get EAs (GEA)
- a list of get EAs (GEAList)
- EAOPs.

These are described in Figure 33-8 on page 33-25.

Full EAs: An FEA is a complete extended attribute; that is, it contains the extended attribute name and value. The name length does not include the trailing NUL. The characters that form the name are legal file name characters.

An FEAList is a list of FEA structures, preceded by a length of the list (including the length itself). FEALists are used for querying, adding, deleting, or changing EAs. They are required input parameters for the function calls that create or set extended attributes (DosSetFileInfo and DosSetPathInfo). They are required output parameters for the function calls that query extended attributes (DosQFileInfo, DosQPathInfo, and DosEnumAttribute). See "Setting Extended Attributes" and "Getting Extended Attributes."

Get EAs: A GEA is an extended attribute name. The name length does not include the trailing NUL.

A GEAList is a list of GEA structures, preceded by a length of the list (including the length itself). GEALists are used for retrieving the values for a particular set of extended attributes. They are required input parameters for the function calls that query extended attributes (DosQFileInfo, DosQPathInfo, and DosEnumAttribute). See "Getting Extended Attributes."

EAOPs: An EAOP is a data structure that consists of a GEAList, an FEAList, and an error. All extended attribute manipulation is performed using this structure. Before calling an extended attribute function, an application must define an EAOP structure, with the GEAList and FEAList appropriately defined. The relevance of the GEAList and FEAList for each function call is described in the *Control Program Programming Reference*.

Note: FEA and GEA data structures include the lengths of EA names and EA values. EA name lengths of zero are illegal and cause error returns on EA function calls. An EA value length of zero has special meaning. Setting an EA with a value length of zero in the FEA data structure causes that attribute to be deleted, if possible. Getting an EA with a value length of zero in the FEA data structure indicates that the attribute is not present.

```
struct FEA
       unsigned char    flags;          /* byte of flags                */
       unsigned char    cbName;         /* length of EA name            */
       unsigned short   cbValue;        /* length of EA value           */
       unsigned char    szName[];       /* ASCII EA name                */
       unsigned char    aValue[];       /* free-format EA value         */
       ;

struct FEAList
       unsigned short   cbList;         /* length of list               */
       struct FEA list[];               /* set of FEAs                  */
       ;

struct GEA
       unsigned char    cbName;         /* length of EA name            */
       unsigned char    szName[];       /* ASCII EA name                */
       ;

struct GEAList
       unsigned short   cbList;         /* length of list               */
       struct GEA list[];               /* set of GEAs                  */
       ;

struct EAOP
       struct GEAList far * fpGEAList;  /* set of GEAs                  */
       struct FEAList far * fpFEAList;  /* set of FEAs                  */
       unsigned short offError;         /* offset of FEA err            */
       ;
```

Figure 33-8. EA data structures

Getting Extended Attributes

EALIST is a sample program provided with the OS/2 Version 1.2 Programming Tools and Information which has the capability of the DIR command, but which also allows a user to view, add, and update 'Category' and 'Description' extended attributes with a file. It also allows the user to browse all extended attributes associated with a file.

To get Category and Description EAs for files, EALIST performs the basic operations described in Figure 33-9 on page 33-26. The source code that performs these operations can be found in the internal subroutine GetDirFileInfo in the source code module EA.C on your samples diskette.

To browse all EAs for a file, EALIST performs the basic operations described in Figure 33-10 on page 33-27. The source code that performs these operations can be found in the internal subroutines:

- InitializeFEAList

in the source code module EA.C on your OS/2 Version 1.2 samples diskette; and in

- fnwpBrowse
- DisplayEAValue

in the source code module EADLGS.C on your OS/2 Version 1.2 samples diskette.

Setting Extended Attributes

To update, or set, an EA for a file, EALIST performs the basic operations described in Figure 33-11 on page 33-28. The source code that performs these operations can be found in the internal subroutine SetDirFileInfo in the source code module EA.C on your samples diskette.

```
Make a GEAList structure with contents 'Category', 'Description'
        (These are the EA names EALIST operates on.  That is, these
        are the names of the EAs for the file whose values will be returned
        by DosQPathInfo.)

Allocate memory for and make an FEAList structure, where FEAs for Category and
        Description EAs associated with files will be returned by
        DosQPathInfo.)

Make an EAOP structure, containing pointers to the GEAList and FEAList
        structures just defined.

Call DosQPathInfo for FileInfoLevel 3 to get EA values for Category and
        Description EAs.  (The FileInfoBuf parameter is, in this case,
        actually the EAOP data structure that has been previously defined.)
```

Figure 33-9. Pseudocode for getting EAs for a file

Manipulating File Objects with EAs

Extended attributes are supported in the OS/2 Version 1.2 FAT file system and installable High Performance File System. Extended attributes are not supported by:

- DOS 3.3/4.0
- OS/2 Version 1.1.

EAs belonging to file objects are lost, therefore, when moving those file objects from a storage device managed by the OS/2 Version 1.2 FAT file system or by an installable file system to a storage device managed by the old FAT file system.

EAs associated with a file object are not part of a file object's data, but are maintained separately and managed by the file system that manages that object. For obvious reasons (primarily removable media), EAs reside on the volume on which the file object resides and are connected to their file object by pointers from the file object. Because of the connectivity of EAs to their file objects, EAs belonging to a file, for example, are also lost when an application saves the file by doing an open/replace of the original file.

Preserving EAs

Programs written for releases of OS/2 and DOS that do not support EAs do not know about EAs, and will, therefore, tend to lose them simply because they do not know that they are supposed to preserve them. While there is no general solution for this problem, OS/2 provides some controls that prevent old programs from destroying critical data without restricting them unduly. This is done by classifying, or marking, programs and extended attributes associated with files.

Programs are classified as:

- Those that recognize EAs.

 These include OS/2 Version 1.2 programs, as well as all programs written for releases of OS/2 and DOS that do not support EAs.

- Those that do not recognize EAs.

EAs associated with files are marked as critical or non-critical. Programs that do not recognize EAs are not allowed to manipulate files that have critical EAs associated with them.

This protection cannot be effectively applied to directories. EAs associated with directories cannot be marked as critical. To forbid programs that do not recognize EAs manipulations on directories that might lose EAs (through tree-copy, backup, or restore) would require that we prohibit them from traversing the directories or enumerating their contents. This is an unacceptable restriction.

Defining Critical EAs: EAs are non-critical by default. A non-critical EA is one that is not necessary to the application's function. That is, if a non-critical EA is somehow lost, the system continues to operate correctly. For example, losing the icons associated with data files does not generally cause any ill effect other than the inability to show the icon.

A critical EA is one which is necessary to the correct operation of the system or a particular

```
For all EAs associated with the file:

Get the name of an EA and the length of its value by calling
    DosEnumAttribute

Make a GEAList data structure with the name of the EA just returned
    from DosEnumAttribute as the contents.

Allocate memory for an FEAList data structure large enough to return
    the value of the EA just returned from DosEnumAttribute.

Make an EAOP structure, containing pointers to the GEAList and FEAList
    structures just defined.

Call DosQPathInfo for FileInfoLevel 3 to get EA values for Category and
    Description EAs.  (The FileInfoBuf parameter is, in this case,
    actually the EAOP data structure that has been previously defined.)

Display the EA name and its value.
```

Figure 33-10. Pseudocode for browsing EAs for a file

application. EAs should be marked as critical if their loss would cause the system to behave incorrectly. For example, mail headers can be stored in EAs. The loss of the header from a message would normally render the mailer completely unable to further use that message. This would be unacceptable, so a mailer should mark its EAs as critical.

Marking EAs as 'critical' only prevents programs that do not recognize EAs from losing the EAs from the file. It does not prevent deletion of files by any application. It also does not prevent programs that recognize EAs from trashing them.

Note: Applications must be wise in how they mark their EAs. If they are not, they will annoy users by preventing some applications from touching files that they could just as well be used with.

Marking EAs: A file has critical EAs if at least one EA attached to the file has the critical bit set.

The critical bit is the 0x80 bit of the flags byte of the FEA data structure. (See Figure 33-8 on page 33-25.) If this bit is 0, the EA is a non-critical EA. If it is 1, it is a critical EA. The constant FEA_NEEDEA found in the BSEDOS.INC and BSEDOS.H include files on the diskettes accompanying the OS/2 Programming Tools and Information can be used to indicate a critical EA.

Recognizing EAs: Programs that do not recognize EAs are restricted from performing certain operations on files that have critical EAs associated with them. For example, a program that does not recognize EAs is not allowed to perform a non-truncating open. An application cannot be allowed to read the data and to forget about the EAs. Programs that do not recognize EAs are, however, allowed to perform those operations which they can do completely. For example, they may delete files with critical EAs associated with them.

Programs that recognize EAs understand EAs. Therefore, no restrictions are placed on their operations. Programs that recognize EAs have no restrictions with respect to critical EAs. They may do what they wish.

The classification of programs according to EA recognition is not a function of DOS mode versus OS/2 mode applications or of previous versions of OS/2 and DOS versus OS/2 Version 1.2. The creator of an EA gets to declare whether it is important (critical) or not. Programs that do not recognize EAs are not prohibited from touching files whose EAs do not matter. See Figure 33-12 on page 33-29.

Marking Programs: A program that uses long file names and recognizes and manipulates EAs must identity itself to the operating system. This is done by including the NEWFILES declaration in the program's module definition file. See "Using Long File Names" on page 33-8.

```
Make an FEAList data structure containing the EA names and values that
    are being set (updated).

Make an EAOP data structure, containing pointers to the FEAList data structure
    just defined and a dummy GEAList data structure.  (The GEAList data
    structure is not required for the function call.)

Call DosOpen to open the file with deny/write access (to control access to
    the file's EAs).

Call DosSetFileInfo for FileInfoLevel 3 to set EAs specified in the FEAList.

Call DosClose to close the file.
```

Figure 33-11. Pseudocode for setting EAs for a file

Moving File Objects with EAs

Files with EAs may lose their EAs when:

- They are transferred to down-level systems (for example, those running OS/2 Versions 1.0 and 1.1).
- They are open for replacement by programs that do not recognize EAs (for example, editors written for DOS and OS/2 Versions 1.0 and 1.1).
- They are sent over COM links.

So that files with EAs can be manipulated under these circumstances, without losing EAs, OS/2 provides a service to separate EAs from their file objects and place them in separate files that the user can work with.

The user may:

- Strip EAs off files he wants to edit with a program that does not recognize EAs.
- Get EAs into normal files so they can be dealt with by old tools (backup, restore, etc.) or easily transmitted.
- Reattach EAs to files brought back from down-level systems where EAs may have been lost or stolen.

OS/2 Version 1.2 offers a utility called **EAUTIL**, which allows users to separate EAs from and rejoin EAs with their file objects. See -- Fig 'EASPLIT' unknown --. This utility allows users to separate extended attributes from the specified file object, optionally disassociating them from the file object. The extended attributes are placed into a specified HoldFile. This utility also allows users to rejoin extended attributes, previously separated from their file objects, with their file objects.

EAUTIL may be applied to subdirectories, as well as to files. It does not support wildcard (meta characters) parameters; it operates on a single file object at each invocation. For more information, see the OS/2 Version 1.2 Online Command Reference.

Searching for File Objects

The following function calls allow applications to search for OS/2 file objects:

DosFindFirst
: Finds the first matching file object

DosFindFirst2
: Finds the first file object with matching EAs

DosFindNext
: Finds the next matching file objects

DosFindClose
: Ends the search for matching file objects

DosScanEnv
: Searches for an environment variable

DosSearchPath
: Searches directory paths for a file object.

Finding a Matching File Object

DosFindFirst can be used with DosFindNext to search along directory paths to find successive matching files.

DosFindFirst returns the name of the first file that matches the file name specified, which can include global file name characters.

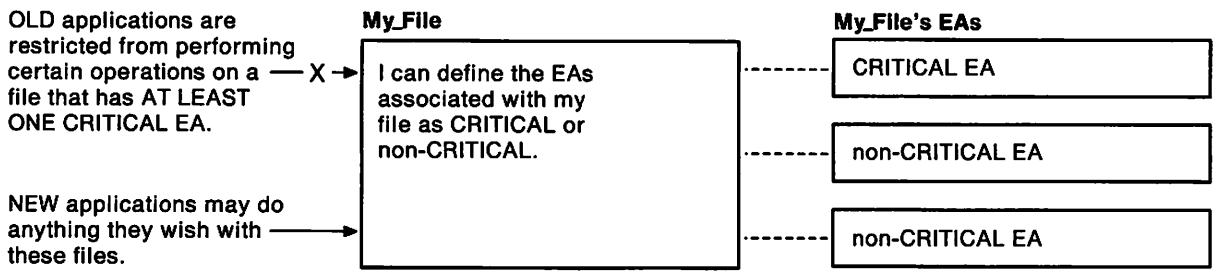


Figure 33-12. Accessing file objects with extended attributes

A request handle is returned to call DosFindFirst. If you do not intend to use this call in conjunction with DosFindNext, you can give the handle a value of 0001H, which is always available to the requesting process. Otherwise, specify FFFFH, which returns a handle at this location. In addition, an application can indirectly look at the EAs for file objects returned by the **DosFindFirst2** function call.

The returned handle can then be used as a search key with **DosFindNext** to find the next file name that matches the file specification. The handle can also be used with another DosFindFirst to find a new file specification. When used with another DosFindFirst, the handle's association with the previous DosFindFirst is ended.

An attribute parameter for DosFindFirst allows hidden and system files, as well as normal files, to be included in searches. The file information returned by a search includes file dates and times, length of data in the file, file size, attributes, and name.

To find all files that match the file specification, issue DosFindNext repeatedly until the message ERROR_NO_MORE_FILES is returned. Then issue **DosFindClose** to close the directory handle. After DosFindClose is issued, a subsequent DosFindNext fails.

Sample Programs: Examples of the **DosFindFirst**, **DosFindNext**, and **DosFindClose** calls are found in the following directories:

```
\TOOLKT12\C\SAMPLES\BSE\EALIST\EA.C
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\C\SAMPLES\PM\IMAGE\IMAGE.C
```

Searching Paths

DosScanEnv searches the environment segment for an environment variable name; for example, DPATH. The result pointer points to the first character of the string that is the value for the environment variable. This call can be made directly by the application, or it can be invoked by DosSearchPath.

DosSearchPath searches directory paths for the name of a file object. The file specification can include global file name characters.

The path string used in the search consists of directory paths separated by semicolons. The caller can supply the path string, or it can supply the name of an environment variable whose value is the path string to be searched. The caller can request that the current working directory be searched before the path string is searched.

If the caller specifies an environment variable, DosSearchPath uses DosScanEnv to find the path string.

If the file is found, its full path name is returned, with global file name characters left in place. The results may not be meaningful if a buffer overflow occurs.

Figure 33-14 on page 33-30 illustrates two methods for searching directory paths to find a file.

Sample Programs: An example of the **DosSearchPath** call is found in the following subdirectory:

```
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
```

```
To edit a file with CRITICAL EAs using an OLD application,
you must first separate the EAs from their file object
by using the EAUTIL utility.
```

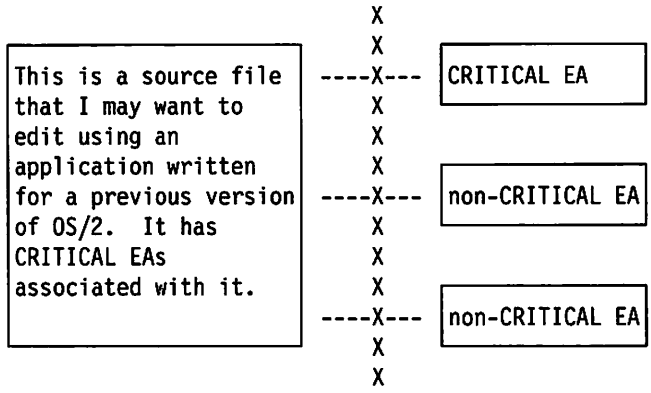


Figure 33-13. OLD applications accessing files with CRITICAL EAs

Searching for File Objects with EAs

An application can search for file objects that have specific EA names by calling **DosFindFirst2** as illustrated in Figure 33-15 on page 33-31.

Reading and Writing Data

After a file has been opened with DosOpen, the read/write pointer is positioned at the first byte of the file. When reading or writing to a file, the file's read/write pointer is advanced to the desired position by issuing read and write calls or by issuing DosChgFilePtr.

If a file is read only, a write to the file will not be performed. The read-only attribute of a file can be changed before it is opened by issuing DosSetFileMode. See "Manipulating File Attributes" on page 33-13.

The following function calls support reading and writing to a file, pipe, or device:

DosRead
: Synchronously reading a file, pipe, or device

DosReadAsync
: Asynchronously reading a file, pipe, or device

DosWrite
: Synchronously writing to a file, pipe, or device

DosWriteAsync
: Asynchronously writing to a file, pipe, or device

DosChgFilePtr
: Moving the read/write pointer to a specified location in a file

String in Environment Segment

```
"dpath=c:\sysdir;c:\init"
```

Method 1, using DosScanEnv and DosSearchPath

```
DosScanEnv("dpath", &PathRef);
DosSearchPath(0,                    /* Path Source Bit = 0 */
    PathRef, "myprog.ini", &ResultBuffer, ResultBufLen);
DosOpen(ResultBuffer, ... );
```

Method 2, using DosSearchPath

```
DosSearchPath(1,                    /* Path Source Bit = 1 */
    "dpath", "myprog.ini", &ResultBuffer, ResultBufLen);
DosOpen(ResultBuffer, ... );
```

Figure 33-14. Searching Directory Paths

DosFileLocks
Locking and unlocking a region in a file to restrict its access

DosFileIO
Performing combined file I/O operations

DosBufReset
Flushes cache buffers to disk.

Reading from a File, Pipe, or Device

Issuing **DosRead** with a handle for a file, pipe, or device transfers a specified number of bytes from the object to a buffer location. The system returns the number of bytes actually read, which may not be the same as the number requested because the end of the file may have been reached.

Issuing **DosReadAsync** with a handle for a file, pipe, or device transfers a specified number of bytes from the object to a buffer location. Transfer takes place while the thread of the calling process continues its execution.

When performing an asynchronous read, the calling process must pass the address of a semaphore it has defined to the system, so the system can notify the process when the read operation is completed. The caller should set the semaphore for signaling purposes by issuing DosSemSet before it issues DosReadAsync, and then issue DosSemWait after it issues DosReadAsync, to wait for the system to clear the semaphore.

The value of the file's read pointer is updated before the read request is queued to the device driver. The calling process should not attempt to look at returned values until the semaphore is cleared.

Note: See "Reading a Named Pipe" on page 35-14 for special considerations pertaining to named pipes.

Sample Programs: Examples of the **DosRead** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\IMAGE\IMAGE.C
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
```

An example of the **DosReadAsync** call can be found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROG2.ASM
```

Writing to a File, Pipe, or Device

Issuing **DosWrite** with a handle for a file, pipe, or device transfers the number of bytes specified from a buffer location to the object. The system returns the number of bytes actually written, which in the case of a disk file may not be the same as the number requested because of insufficient disk space.

Issuing **DosWriteAsync** with a handle for a file, pipe, or device transfers the number of bytes specified from a buffer location to the object. Transfer takes place asynchronously with the caller's execution.

Before issuing an asynchronous write, the calling process must define a RAM semaphore, which is used by the system to signal the process when the write operation is completed. The address of the semaphore is passed with DosWriteAsync to the system. The caller should set the semaphore for signaling purposes by issuing DosSemSet before it issues DosWriteAsync, and then wait for notification by the system after calling DosWriteAsync by issuing DosSemWait.

The value of the file's write pointer is updated before the write request is queued to the device driver. The calling process should not attempt to look at returned values until the semaphore is cleared.

If an application requires data to be written in a specific order, setting the Write-Through flag to 1 guarantees that actual I/O for a synchronous write is completed before the DosWrite returns. If this flag has been set with DosOpen for buffered I/O, and multiple synchronous writes are issued, the system cannot guarantee the actual order in which the data is written. To change the state of the Write-Through flag, see "Determining the State of an Open File or Device" on page 33-14.

```
Call DosFindFirst2 for FileInfoLevel = 2, to get the length of the buffer
    required to hold the EAOP data associated with a matching file object.

Call DosFindFirst2 for FileInfoLevel = 3, to get the EAOP data associated with
    the matching file object.
```

Figure 33-15. Pseudocode for searching for files with EAs

Note: See "Writing to a Named Pipe" on page 35-14 for special considerations pertaining to named pipes.

Sample Programs: Examples of the **DosWrite** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Moving the Read/Write Pointer

The read/write pointer of a file is advanced a specified distance in bytes by issuing **DosChgFilePtr**. The location from which the pointer is moved can be the beginning of the file, its current location in the file, or the end of the file.

Moving the pointer from the end of the file can be used to determine the file's size.

Sample Programs: An example of the **DosChgFilePtr** call is found in the following subdirectory:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
```

Controlling Access to a Region within a File

DosFileLocks provides a simple mechanism that temporarily prevents access by other process' to regions within a file. File locks are intended to be in effect for only short periods of time.

Locking and unlocking regions in a file enables sharing processes to coordinate their efforts. A process can lock a region in a file so it can read and update it. A sharing process that attempts to lock the region before the other process finishes its update and unlocks the region receives an error code. When a lock is unsuccessful because a lock is already in place, ERROR_LOCK_VIOLATION is returned.

A lock that extends beyond end-of-file is not considered an error. However, a locked region cannot overlap another locked region, nor can it encompass a locked region. Any such conflicting locks must be cleared before a region can be locked.

When a file handle is duplicated, the duplicate handle has access to any locked regions currently being accessed by the original handle. Although a child process created with DosExecPgm can inherit the file handles of its parent process, it does not inherit access to locked regions in effect for a file handle unless the file handle is duplicated and passed to it.

When a file with locks is closed, the locks are released in no defined order.

Performing combined file I/O

DosFileIO provides a combined method of locking and unlocking multiple file ranges, changing the position of a file pointer, and performing read and write operations on a file.

This function provides a simple mechanism for excluding other processes' read/write or write access to regions of the file. If another process attempts to read or write a *no-access* region, or attempts to write in a *read only* region, the system call will return an error indicating exclusion to that region.

Note a range to be locked must first be cleared of any locked subranges or locked overlapping ranges.

Flushing Cache Buffers to Disk

The **DosBufReset** call is used to flush the file system's cache buffers for a specific file handle. When issued with a value of FFFFH for the file handle, DosBufReset writes out all files belonging to the requesting process. This usage should be administered with care, so the user is not burdened with insertion and removal of a large number of removable media volumes.

When DosBufReset is issued for a file handle, the directory entry for the file is updated as if the file had been closed. However, the file remains in an open state.

DosBufReset may also be issued with the name of a named pipe. The process that issues DosBufReset is blocked at one end of the pipe until all data it has written has been successfully read by the process at the other end of the pipe. This allows communicating processes to synchronize their dialogs. For information on using named pipes, see "Two-Way Communication with Named Pipes" on page 35-11.

Chapter 34. Program Execution and Control

This chapter describes the calls associated with the multitasking hierarchy of threads, processes, and sessions. To promote easy to write and easy to run OS/2 applications, timer services, dynamic linking, and debugging are also included in this chapter.

The OS/2 1.2 Programming Tools do not provide sample programs for all function calls discussed in this chapter. However, when an example of a function call is used in a sample program, the subdirectory path of the sample program is shown. **Note that a sample program may not illustrate the task of the function call as discussed in the chapter text.**

Threads and Processes

There are many possibilities for the design of an application, using processes and threads.

A process is a collection of resources that includes code and data in an executable file and any system resources called by the code, such as system semaphores, allocated memory, files, pipes, and queues. A thread is a dispatchable unit of execution that uses the resources of the process.

A process can start other processes, and within a process one or more additional threads can be started. The creation and termination of a process is relatively slow, compared to the creation and termination of a thread.

Starting a process is more costly to system resources than starting another thread of execution within the process. Sharing data and resources between processes requires shared memory and the mechanisms of interprocess communication.

A simple Presentation Manager application consists of a single process and will usually have two threads:

- A user interface thread, which listens and responds to user requests and queues work for the second thread.
- A processing thread, which handles any lengthy processing.

For some suggestions on designing a Presentation Manager application, see "Multithreading Considerations for PM Applications" on page 34-4.

Figure 34-1 on page 34-2 illustrates a multiple threads solution for an asynchronous execution requirement. Each thread uses a different device to complete a portion of the overall solution.

Starting Another Thread of Execution

OS/2 creates the first thread of execution for a process when it starts the executable file. To create another thread of execution, a thread allocates stack memory and then issues **DosCreateThread**, specifying the address within the program module where execution is to be started. Upon creation, the new thread is dispatched to the address and begins asynchronous execution.

The started thread has a unique stack and register context and shares the resources of the process with the other threads in the process.

Note the minimum available space on the stack for a thread calling an OS/2 function is 4KB. The thread also has a priority, which is inherited from the thread that created it. To change a thread's priority, see "Setting Priorities for Threads and Child Processes" on page 34-10.

Code segments execute in one of three privilege levels: the operating system and device drivers at privilege level 0 (the highest level); special purpose I/O routines such as presentation drivers at privilege level 2; application programs at privilege level 3. All code segments executing at privilege level 3 that have IOPL can access data executing at privilege level 2 by calling **DosCallBack**. **DosR2StackRealloc** cannot be called from privilege level 2. However, DosR2StackRealloc is used by a privilege level 3 routine before calling privilege level 2 to ensure that privilege level 2's stack is large enough. Note that WinInitialize (WMINIT) uses DosR2StackRealloc.

DosCreateThread cannot be issued from within a segment that has I/O privilege. When this is done, the DosCreateThread request causes a general protection fault and the process is terminated. If it is necessary to create another thread within an IOPL segment, DosCallBack can be used to invoke a

privilege level 3 segment, which issues the DosCreateThread request.

For more information on IOPL code segments and I/O privilege levels, see *I/O Subsystems and Device Support: Volume 1.*

Sample Programs: Examples of the **DosCreateThread** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC1\GRAPHIC1.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\MASM\SAMPLES\BSE\THREADB\THREAD.ASM
```

Suspending Threads within a Process

Threads can suspend the execution of other threads within the same process and thereby control access to serially reusable resources of the process. The following function calls support this activity:

DosSuspendThread
Suspends execution of a thread

DosResumeThread
Resumes execution of the suspended thread

DosEnterCritSec
Suspends execution of all other threads

DosExitCritSec
Resumes execution of suspended threads.

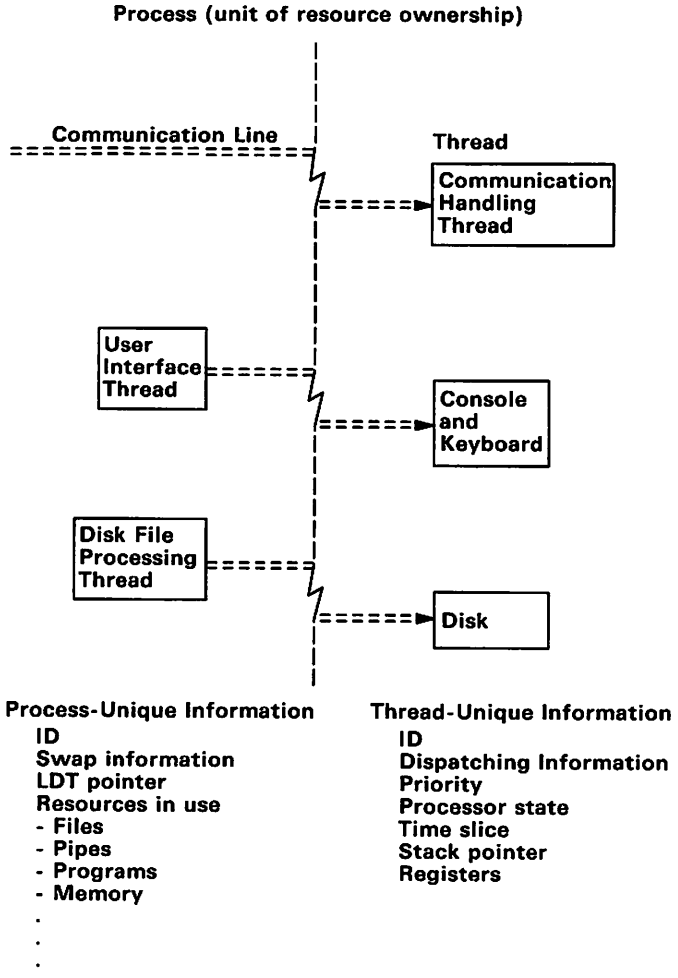


Figure 34-1. Multiple Threads within a Process

Suspends Execution of a Thread

A thread's execution is suspended when another thread in its process issues **DosSuspendThread** with the target thread's ID.

The thread may not be suspended immediately because it may have locked system resources that should be freed first. However, the thread blocks itself from executing any further application instructions until a **DosResumeThread** is issued.

Suspending Execution of All Other Threads

A thread can suspend execution of all other threads in its process by issuing **DosEnterCritSec**. This call causes the other threads to block themselves and give up their time slices, thus enabling the calling thread to access a time-critical resource of the process.

With the exception of thread 1 (the initial thread of execution), which has been dedicated to signal handling, execution of the suspended threads will not resume until the active thread issues a **DosExitCritSec**. If a signal occurs while a DosEnterCritSec is active, thread 1 can execute to process the signal. Of course, thread 1 should not access the resource being used by the thread that issued DosEnterCritSec. (For a discussion of signal handling, see "Setting Flags to Respond to Time-Critical Events" on page 35-6.)

A count is made of outstanding DosEnterCritSec requests. The count is incremented when a DosEnterCritSec is made and decremented when a DosExitCritSec is made. A DosExitCritSec causes normal thread dispatching to resume when the outstanding DosEnterCritSec count is 0. This count is maintained in a word. If overflow occurs, the count is set to a maximum value and an error is returned.

Threads that contain code that is time-critical to other threads in the process should not make dynamic link calls within these critical sections. The dynamic link procedure can be serializing a resource, using semaphores. If a thread entering the critical section blocks another thread already owning the resource about to be requested by the dynamic link function, a deadlock occurs. As an example, suppose threads of an application are serializing their access to a queue by means of a semaphore. A thread enters a critical section and issues a request to read the queue while another thread already has the semaphore that controls access to the queue. The thread that has the semaphore is now effectively blocked by the DosEnterCritSec call, and the thread that has requested the queue waits forever to access it.

Ending the Current Thread or Process

To end the execution of the current thread, call **DosExit**, specifying the action code as 0. When DosExit is specified with the initial thread of execution started by OS/2, the value of the action code is ignored, all threads are terminated, and the process is ended.

It is good practice to end each thread in your application individually.

Figure 34-2 on page 34-4, taken from the MEMORYC.C sample program in the TOOLKT12\C\SAMPLES\BSE\MEMORYCB subdirectory, illustrates ending a thread.

Sample Programs: Examples of the **DosExit** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\CRERRCB\CRERRC.C
\TOOLKT12\C\SAMPLES\BSE\DLCBAT\DYNLINKC.C
\TOOLKT12\C\SAMPLES\BSE\HELLOCB\HELLOC.C
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSEC.C


\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\BMAP\BMAP.C
\TOOLKT12\C\SAMPLES\PM\CLIPBRD\CLIPBRD.C
\TOOLKT12\C\SAMPLES\PM\DIALOG2\DIALOG2.C


\TOOLKT12\MASM\SAMPLES\BSE\DEMODB\MONITOR.ASM
\TOOLKT12\MASM\SAMPLES\BSE\DYNBLD\DYNLINK.ASM
\TOOLKT12\MASM\SAMPLES\BSE\FAMAPI\VIOSAMP.ASM
\TOOLKT12\MASM\SAMPLES\BSE\HELLOBAT\HELLOWRD.ASM
```

Variable used by DosExit

```
#define  ACTION        0          /* end the current thread for DosExit     */

   unsigned result = 0;            /* contains the result code upon exit     */
```

The call to DosExit to end a thread

```
   DosExit(ACTION, result);
```

Figure 34-2. Ending a Thread's Execution

Multithreading Considerations for PM Applications

This section gives you some hints and tips for designing a multithreaded PM application.

A thread that creates a window must have an input message queue. Only the thread that creates a window can receive user input to the window. All windows of an application should be created in the user interface thread that creates the input message queue, unless the design of your application dictates otherwise. Doing anything else can give poor performance.

You can, however, create one or more asynchronous threads to perform any lengthy processing needed by a window. The user interface thread passes requests for lengthy processing on to an asynchronous thread. This leaves the user interface thread free to process its message queue, and so remain responsive to the user. For this arrangement to work efficiently and in a well-ordered manner, the threads must coordinate their processing.

For example, a simple multitasking PM application uses a single process with two threads:

1. A user interface thread, listening to user requests, and queueing any work that is for the other thread.
2. A processing thread that handles any potentially lengthy processing.

The user interface thread creates an input queue, creates and owns the windows seen by the user, and creates the processing thread. The user interface thread uses WinGetMsg and its window procedure in the normal way to interpret the user's input. If the user's input requires work that involves lengthy processing, the work request is communicated to the processing request. If necessary, the user interface thread can communicate a request that the processing thread stop.

While the processing thread is executing, the user interface thread has various options for staying responsive to the user, depending on the nature of the function the application is providing:

- Stop processing its current task and do what the user has just requested.
- Allow the user to interact with other parts of the application. If this is not practical, it should display a message saying that it is processing the user's request and cannot do anything else for the moment.
- Offer the user the option of switching away from the application and bringing another application into the foreground.
- Offer the user the option of terminating the application completely.
- Queue the request and do it later.

The processing thread listens for work requests generated by the user interface thread. When work requests are received, the processing thread processes the requests in sequence.

Ideally, the processing thread regularly listens for requests to stop its processing, even in the middle of work, if practical. If a stop request is received and it is practical to stop work, the processing thread should stop the work it is doing, tidy up as necessary, and proceed to the next request. Whether the processing thread returns to the stopped work can depend on the nature of the next request. For example, if the program is a graphics program, the processing thread is in the middle of drawing a picture, and the user requests a sizing operation, there is no point in the processing thread returning to its stopped work and wasting its time drawing, when it will only have to redraw a sized picture later.

If it is not practical to stop work, the processing thread should communicate this to the user interface thread, finish the work it is doing, and then proceed to the next request.

The processing thread should also listen for requests to terminate the application, and should shut itself down in an orderly way, and notify the user interface thread.

In the above example, coordination between the threads is achieved by communication between the threads. There is a variety of ways that a multithreaded PM application can use to achieve the communication:

- RAM semaphores, described in "RAM Semaphores" on page 35-2
- Global variables used as flags.
- PM Messages. A user interface thread and a processing thread can post each other messages. A processing thread that uses PM services must create its own anchor block, using the WinInitialize call. A processing thread that receives messages must create its own message queue and have a procedure to handle the messages.
- Shared data. For non-PM data objects, if you use a message to pass a pointer to some data to another process, you must make sure that the data is addressable by that process, and make sure that there is a protocol for freeing the storage at the end.
- A process can use the WinLockWindow call to lock and unlock a window of another thread, This ensures that the window is not destroyed while the first process is using it.

This section gives you examples of how to use some of the above facilities.

Example – Processing Thread Without a Message Queue

In this example, there is:

- A user interface thread, with an input message queue
- A processing thread without a message queue.

The user interface thread uses semaphores to communicate with the processing thread. The user interface thread cannot send or post messages to the processing thread, because the processing thread has no message queue.

Because the processing thread has no message queue, the calls that it can use are limited. Most output functions (Gpi, Vio, Dev) are allowed. This means that drawing can be done on a thread that has no queue, even though the window to which the data is drawn belongs to a different thread. However, see the next section for the considerations of multiple threads accessing a single presentation space. See the *Control Program Programming Reference* for details of the calls that work without a message queue. Without a message queue, a thread is limited in the Win calls it can use. It cannot explicitly send messages to windows, or use calls that implicitly send messages. However, posting messages to a window or another input queue is allowed. So, the processing thread uses messages to communicate with the user interface thread.

In this example, the user interface thread wants to be able communicate three events to the processing thread:

1. When to start its work
2. When to interrupt its work
3. When to terminate its work completely.

To achieve the communication, the application sets up a RAM semaphore hsemStart, and two global variables fPause, and fTerminate.

The processing thread wants to be able to listen for the above events, and communicate the following three events to the user interface thread:

1. It has started its work
2. It has completed its work
3. It has terminated its work completely.

To achieve this, the application defines its own application-internal messages:

```
#define ME_WORKING    WM_USER
#define ME_COMPLETED  WM_USER + 1
#define ME_TERMINATED WM_USER + 2
```

The user interface thread uses the semaphore hsemStart to start the thread working. It uses the two flags fPause and fTerminate, initially set to FALSE, to communicate to the processing thread whether it can continue, and whether it should terminate. If the user interface thread wants the processing to pause, it sets fPause. If it wants it to terminate, it sets fPause and fTerminate.

The processing thread waits on hsemStart to tell it to start. When it starts, it posts a ME_WORKING message to the user interface thread. In its work loop, it checks the status of fPause. If fPause is TRUE, the processing loop saves what it is doing, and breaks out of its work loop. Outside the work loop, if fTerminate is TRUE, it tidies up, posts a ME_TERMINATED message to the user interface thread, and exits the processing thread. If fPause is FALSE, the work must be complete, so it posts a ME_COMPLETED to the user interface thread. The processing thread returns at this point.

Example – Processing Thread With a Message Queue

In this example, the processing thread sets up its own message queue. The user interface thread and the processing thread communicate in both directions by posting the following application-defined messages to each other:

```
/* User thread to processing thread */
#define YOU_WORK      WM_USER
#define YOU_PAUSE     WM_USER + 1
#define YOU_TERMINATE WM_USER + 2

/* Processing thread to user thread */
#define ME_WORKING    WM_USER + 3
#define ME_COMPLETED  WM_USER + 4
#define ME_TERMINATED WM_USER + 5
```

The communication is illustrated in Figure 34-3.

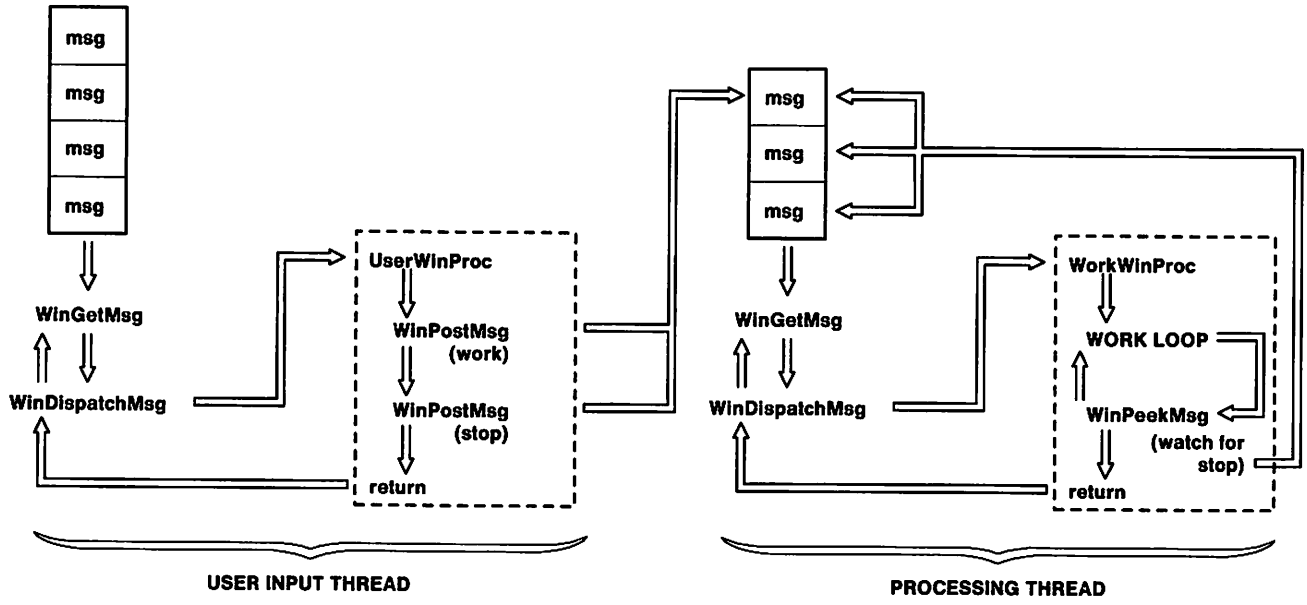


Figure 34-3. Interthread Communication Using Messages

The user interface thread creates the processing thread. The processing thread initializes PM, creates its own message queue, registers and creates an object window, and has a window procedure.

As a result of receiving the normal PM-defined input messages, the user interface thread posts the application-defined messages YOU_WORK, YOU_PAUSE, or YOU_TERMINATE to the object window.

The processing thread message queue receives only messages posted from the user interface thread. The processing thread gets YOU_WORK messages from its queue and dispatches them to its window procedure. Inside the processing thread's work loop, it uses WinPeekMsg (WMPEEK) to listen for YOU_PAUSE and YOU_TERMINATE.

The sample programs GRAPHIC1 and GRAPHIC2 work in a similar way to the above example, except that:

- The processing thread does not create an object window. This means that the user interface thread has to use WinPostQueueMsg (WCPQM) to post messages to the processing thread's queue, instead of using WinPostMsg (WMPOST) to post messages to its window.
- The programs also use flags protected by a semaphore, to coordinate when the processing thread can be interrupted by the user interface thread.

Multiple Threads Painting the Same Presentation Space

In a single process, you may want multiple threads to be able to paint data into a single presentation space. In this case, your application is responsible for serializing access of the presentation space by the threads. Semaphores can be used for this purpose. You cannot serialize on the PS busy signal returned by the GPI. It exists only for the protection of the system.

A presentation space is owned by the process that creates it. A process cannot access a presentation space that belongs to another process. The only way for a process to affect the presentation space of another process is for the first process to send messages to the second process's window as described below.

Object Ownership

Various presentation management programming objects are created through the use of the PM API. The objects have handles, and include windows, presentation spaces, message queues, device contexts, window classes, pointers, bit maps, accelerator tables and metafiles. Except for metafiles, sharing of handles among processes is not allowed for the majority of objects. A metafile handle can be passed from one process to another for use by the second process. See the *Control Program Programming Reference* for full details.

The objects can be destroyed explicitly by function calls. However, the general rule is that objects are destroyed when the process that created them is destroyed.

If a process uses an object which it did not create, there is always a possibility that the object may be destroyed while the process is using it. This can happen even if the process is cooperating with the creating process, as the creating process could die due to a severe error.

The general rules for object ownership are as follows:

1. Object sharing is not allowed among application programs except for objects passed through the clipboard.
2. An object created by a process is *owned* by the process. If a process attempts to access an object owned by another process, an error is returned by the accessing call. When an owning process terminates, the owned object is deleted.
3. An object passed to the clipboard becomes the property of the system and is not deleted when the process that created it terminates.
4. Access to an object in the clipboard can be serialized by the WinOpenClipboard (WIOPCB) and WinCloseClipboard (WICCBD) calls. The process reading the clipboard does not become the owner of the object in it. The process can place an object in the clipboard. The process should not attempt write on or free the object in the clipboard, but this is not enforced by the API.

Owner Windows

Any window can own any other window, providing both windows belong to the same thread. There can be no window ownership across threads.

Scheduling

Use of DosSleep to Relinquish Timeslice

The DosSleep call suspends a task for a specified time interval. If used, it should be used with care. For example, DosSleep(0) can monopolize the processor by not allowing threads of a lower priority to execute. If you must use DosSleep, set the time parameter high to avoid large wastage of processor time.

Priority of Asynchronous Processing Threads

All threads that own windows have the same maximum normal priority. The effect of this is that, if you have a thread that is not well-behaved and consumes the processor for a long period of time without issuing WinGetMsg (WMGET), it will hold up the system.

When you are using an asynchronous thread to perform some processing for a window, the general rules are:

- If the processing is going to make intensive use of the processor, set the thread priority low.
- If the processing is I/O related, set the priority high.

Non-Reliance on Timing of Messages

Do not expect any message to form part of a sequence or to arrive at a particular time. Your application should process each message in its own right.

Shared Data

There are no PM considerations for shared data. The API functions provided for shared data are in the base part of the product, and are described in Chapter 32, "Memory Management."

Private Data

How applications maintain private data depends on whether the scope of the data applies to the process, thread, or window, as described in the subsections that follow.

Process Private Data

If you are writing a program that is going to be in an .EXE file and not use multiple threads, you can use data in static storage that is purely local to your process.

Thread Private Data

If you are writing an .EXE file that uses multiple threads, each thread gets the same copy of the static data, so there is a potential serialization problem. There are two ways round this:

1. Use automatic storage allocated on the stack.
2. Have a table in static storage that contains a thread ID. From that table you find the address of some private data associated with the thread.

If you are writing a .DLL, it is possible for you to get problems between processes because the automatic data segment of a .DLL module can be global. You would then need to use the second technique to maintain process or thread local storage.

Window Private Data

You can use the window words of a window to implement window private data.

Window Handling

Clipping at Parent and Sibling levels

To avoid continual redraw at sibling level, let the default window procedure deal with the WM_ERASE_BACKGROUND message. It is simpler to use WinFillRect (WIFRCT) in your WM_PAINT message processing to redraw the background.

A parent window should clip its children, and each child should clip its siblings.

Shutting Down Multithreaded Multiwindowed Applications

Windows owned by WC_FRAME windows or windows owned by children of WC_FRAME windows are destroyed automatically by the WC_FRAME window processing of the WM_DESTROY window message. The normal processing of WinDestroyWindow (WIDEL) does not destroy owned windows; it is the responsibility of the window being destroyed to resolve these problems.

If a non-WC_FRAME window that owns another is destroyed, the reference to the destroyed window still exists in the owned window: it is the responsibility of the application to break the ownership relationship in this case. This can be done in the WM_DESTROY message handling of the owner window, or the owned window's owner may be set to NULL with WinSetWindowULong (WISWUL) before the owner is destroyed.

Starting Another Process

Issuing **DosExecPgm** creates a process environment from an executable file. The process can run as a child process in the current session or as an independent process in the background. To qualify to run asynchronously as an independent process, an application cannot receive input from the keyboard or send output to the screen, except when a Vio pop-up is active.

When a child process is started, it inherits handles to unnamed pipes and files. However, the child has access to only those files where inheritance has been indicated by the parent with DosOpen.

The parent also controls the meanings for standard input, output, and error. For an explanation of how the parent accomplishes this control, see "Creating a Duplicate Handle" on page 33-14.

At the time it issues DosExecPgm, the parent determines the environment strings of the child process. It can specify new strings or allow the child to inherit the parent's strings.

Note: Environment strings contain values for system environment variables, such as PATH, and any variables defined by issuing the SET command. These strings reside in the environment segment of the application. Issuing **DosGetEnv** returns a selector to the environment segment of a process and the offset to the command line within the segment.

The executable file of the child process can be started synchronously or asynchronously. If the child is started synchronously, the parent waits for its completion. If it is started asynchronously, the parent can be notified of its completion by having one of its threads issue DosCWait.

Sample Programs: An example of the **DosExecPgm** call is found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Waiting for an Asynchronous Process to End

If a parent process has specified ExecFlags=2 with DosExecPgm, it can have one of its threads issue **DosCWait** to wait for the completion of an asynchronous child process while other threads of the parent continue processing. When the process ends, its process ID and termination code are returned.

If the child has started another process, DosCWait waits for the completion of any grandchild processes before returning but does not report their status. If the specified child process has multiple threads, DosCWait returns the result code of the last DosExit request.

To wait for the completion of all child processes started by the parent, issue DosCWait with ActionCode=1 and ProcessID=0 repeatedly, until NO_CHILD_PROCESS_EXISTS is returned.

Ending a Process

A process can end itself by issuing **DosExit** with the action code parameter set to 1, which means that all threads of the process are terminated. However, when a process has multiple threads, it is good programming practice to end each thread individually by issuing DosExit with the action code set to 0. Be aware, though, that when DosExit is issued for thread 1 of a process, the action code is ignored and all threads of the process are automatically terminated.

A parent process can end a child process by issuing **DosKillProcess** and specifying its process ID.

Sample Programs: An example of the **DosKillProcess** call is found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Performing Cleanup Operations When a Process Ends

DosExitList maintains a list of routines that are given control when the current process ends, normally or abnormally.

When a process ends, all threads are destroyed, except the one performing the exit list routines. The routines are given control in ascending sequence, according to the calling order value specified by function high-order byte. For example, low (0) will be first with high (0FFH) being last.

If there is more than one routine with the same calling order value, the routines will be executed in LIFO sequence. In other words, the last routine to be declared as an exit list routine of a given order is executed first.

When the first exit list routine receives control, the parameter on the stack, located at SS:SP+4, contains an indicator of the reason for the process' termination. The values of these indicators have the same meanings as the termination codes on a DosCWait or a DosExecPgm request:

0 Normal exit
1 Hard error abort
2 Trap operation
3 Unintercepted DosKillProcess.

Ownership of all exclusive system semaphores is transferred to the thread performing exit list processing. Thus, the thread can request any serialization semaphores without danger of being blocked.

Exit list routines should be designed so they do not make requests to function calls that are higher in the calling order hierarchy.

To ensure timely response to the user, there should be minimum delay in the completion of termination. Thus, exit list routines must be as short as possible.

Setting Priorities for Threads and Child Processes

The following function calls support establishing priorities by a process for one of its threads or for a child process:

DosSetPrty
Changes the priority of process, a descendant, or a thread

DosGetPrty
Determines the current priority of a process, a descendant, or a thread.

The OS/2 scheduler has a concept of priority classes and levels. Within each class, a thread's priority level may vary either through system action or by means of DosSetPrty.

When a thread is created, it is initially dispatched with the same priority as the thread that created it. **DosSetPrty** allows a process to change the priority of:

- A single thread of the current process.
- All the threads of any process.
- All the threads of the current process or a child process, as well as all the threads of any descendants of the current process.

The values that can be specified with DosSetPrty for a thread's priority class are:

0 No change to priority class
1 Idle time
2 Regular
3 Time-critical
4 Fixed-high.

An idle-time thread has the lowest priority and executes only when there are no time-critical, fixed-high, or regular threads to execute.

A time-critical thread has the highest priority and executes before any fixed-high, regular, or idle-time threads.

A fixed-high thread's priority is higher than that of a regular thread and lower than that of a time-critical thread.

For each of the priority classes, there are 32 distinct priority levels, 0 through 31. If a level is not specified, it will default to 0.

Idle-time, time-critical, and fixed-high threads have static priority levels that are not varied by OS/2. They are scheduled among those of their own class in priority order, with round-robin scheduling of threads of equal priority levels.

The majority of threads executing in the system are regular class, because for many applications priority is not a concern. The priority of the initial thread of execution created by OS/2 when it starts an executable file is regular class.

The priority level of a regular class thread is varied by OS/2, based on the thread's actions, such as recent I/O and processor time usage history, and the overall system activity.

To meet a time-critical requirement, you can change the priority of the thread during its critical processing, or you can maintain a separate thread for this function, which is initialized to a time-critical class.

The fixed-high class assists in the synchronization of a background thread that is expected to run at a given priority in relation to a foreground thread. Note that a foreground thread is a thread which runs in a foreground session; a background thread runs in a background session.

If none of the threads in the system alter their priority from the default, then the only priority that exists is that of the thread in the foreground over those in the background.

Note: Your application will have to share the system with others that have time-critical requirements. Therefore, do not arbitrarily choose a high-level priority.

Sample Programs: Examples of the **DosGetPrty** and **DosSetPrty** calls are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\GRAPHIC1\GRAPHIC1.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROG1.ASM
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROG2.ASM
```

Determining the Process ID (PID)

The following calls assist a process in changing a priority level by specifying the Process ID (PID):

DosGetPID
: Obtains the current process' process ID, the current thread's ID, and the process ID of the parent

DosGetPPID
: Obtains the parent's process ID for any specified process.

Starting Another Session

A session can be thought of as a logical console, consisting of buffers for the screen, keyboard, and mouse.

An application may start another process in a separate session. This can be done when the application is not going to manage any I/O done by the process. For example, an application that needs to start another application can start it in a separate session.

There are three types of sessions that can be started: full-screen, text window, and Presentation Manager.

The following function calls support starting a session and controlling child sessions:

DosStartSession
: Starts an independent or a child session

DosSelectSession
: Switches a child session to the foreground

DosSetSession
: Sets user selectability of a child session

DosStopSession
: Ends one or all child sessions.

To issue **DosStartSession**, you must create a data structure that defines the session to be started. Different lengths for the data structure are supported to provide compatibility and various levels of application control.

Application type and length are independent. A Presentation Manager application can be started with a length of 24, and a full-screen application can be started with a length of 50. When an application is started with a length of 24 bytes, the session manager initializes the remaining bytes to 0. This allows the Task List to provide values for missing information, based on the installation file entry for the application being started.

DosQAppType is called by the Presentation Manager shell to determine the application type of an executable file. The application type is specified at link time in the module definition file.

Suppose one application wants to establish an environment for another application but doesn't care whether the application starts in a window or as a full-screen session. In a case like this, length 30 is sufficient. Using length 50 allows the application to control all the current windowing characteristics of the new session.

Any of the three application types can start a session for any other application type. An application's type can be determined by issuing DosGetInfoSeg and testing the value of the process type in the application's local information segment.

Specifying Program Handles

If a process issues a DosStartSession specifying only the program handle, then it must change to the working directory before it issues DosStartSession. The process must be started inherited. If a process is started non-inherited, it will be up to that process to change to the correct directory.

Starting a Child Session

A session may be started as an independent session or as a child session. When a session is started as a child session, the session ID returned by DosStartSession may be used by the process with the rest of the session management calls to manipulate the child session.

There is no parent/child relationship between the process that started the child session and the process in the child session. The true parent of the process in the child session is the Task List When the TraceOpt parameter is set to 2 for use by a debugger, the debugger can obtain the session and process IDs of the true parent for debugging purposes. See "Debugging Aid to Program Authors" on page 34-17.

After a process has started a child session, no other process in its session can start a child session until all dependent sessions started by this process have ended.

Notifying a Process When a Child Session Ends

A process that has started a child session and is attempting to terminate it with DosStopSession will receive a normal return code even when the process that runs in the session refuses to terminate. To ensure that a child session has actually been terminated, the process can be notified by the session manager of child session termination.

The means of notification is a queue that the process creates with DosCreateQueue and names in the TermQ parameter for DosStartSession. The process waits for the session manager to write an element to the queue, signaling the termination of the child session.

The queue is posted, regardless of what causes the termination (user action or a call issued by the process that started the session) and whether the termination is normal or abnormal.

To read the queue, the process issues DosReadQueue with the NoWait parameter set to 0. The word returned in the request parameter of DosReadQueue has a value of 0.

The format of the data element represented by the queue element is:

Size	Description
WORD	Session ID
WORD	Result code.

After reading and processing the data element, the process must free the segment containing the data element by issuing DosFreeSeg.

After all child sessions started by the process have been terminated, the session manager closes the queue.

This same queuing mechanism may also be used by the session manager to notify a debugger when a child session has been started.

Requesting the Foreground

A session may be started as a foreground or a background session. To request a foreground session, the caller's session, or one of its descendants, must be executing in the foreground. Otherwise, the request for a foreground session is overridden; the session is started in the background, and a unique error is returned.

Note: The foreground session for Presentation Manager applications owns the input focus.

When a session is started for a full-screen or text-windowed application, OS/2 adds an entry for the application to the switch list. However, when a session is started for a Presentation Manager application, the application must add the entry.

Controlling Child Sessions

Once a process has started a child session, it can use the session ID returned with DosStartSession and the remaining session management function calls to control the child session. The session ID of a grandchild may not be specified with these calls.

Switching a Session to the Foreground

A process may issue **DosSelectSession** with the session ID of a child session to switch the child to the foreground. At the time of the call, the process' session, or one of its descendant sessions, must be running in the foreground.

The process can also issue this call with its own session ID to switch itself to the foreground when one of its descendants is running in the foreground. This capability is useful to debuggers.

Setting User Selectability

Issuing **DosSetSession** enables a process to affect the selectability of a child session by the user.

A child session can be set as unselectable. The user will be unable to select it from the Task List switch list. If the session is for a Presentation Manager application, the user will still be able to select it with a mouse.

A child session can also be set as bound to its parent session. When the user selects the parent, the child is also brought to the foreground.

Ending Child Sessions

Issuing **DosStopSession** terminates a child session. If the child is running in the foreground at the time this call is made, the parent session is brought to the foreground.

If the process in the session referenced by this call has not ended, DosStopSession still returns a normal return code. You can ensure that a process in a session has ended by waiting for notification from the termination queue specified with DosStartSession.

Notifying the Device Driver of Screen Changes

DosSMRegisterDD allows a device driver to register itself with the session manager.

Device drivers can use the category 11 IOCtl, function 41H to receive notification of all full-screen session screen switches and full-screen, Presentation Manager and text windowed session terminations.

The session manager issues a notification IOCtl to the device driver which has the following format:

Size	Description
WORD	DosDevIOCtl packet length (includes length itself)
WORD	Screen Switch Notification type
WORD	InComing Screen Group number
WORD	OutGoing Screen Group number.

Session Function Call Restrictions

The session manager interfaces described in this section may not be issued under the following conditions:

- If the calling process is started with RUN in CONFIG.SYS.
- If the calling process is started with the DETACH command.
- During a VIO pop-up (by a process that has issued VioPopUp and not yet issued VioEndPopUp).

Session manager calls that change the foreground session are blocked during hard error pop-ups. DosSelectSession only blocks during a Vio pop-up when it is determined the call will be successful when the Vio pop-up ends.

Timer Services

OS/2 provides a full range of interval timers that are useful for synchronizing threads of execution:

DosSleep
: Starts a synchronous timer

DosTimerAsync
: Starts an asynchronous one-shot timer

DosTimerStart
: Starts an asynchronous recurring timer

DosTimerStop
: Stops an asynchronous one-shot or recurring timer

DosGetDateTime
: Gets the current date and time

DosSetDateTime
: Sets the current date and time.

Under optimum conditions, the range of accuracy for these timing functions is between one or two clock ticks because of two factors: the rounding up of time specified in milliseconds to the next clock tick and the system delay in resuming the application. To determine the duration of the clock tick,

issue **DosGetInfoSeg** to examine the timer interval field of the Global Information Segment.

By examining the milliseconds field of the Global Information Segment before and after the timer wait, you can get an approximation of the actual time that has elapsed. Again, the result will be accurate only to between one and two clock ticks. The value in the milliseconds field rolls over every few weeks, and an application may be suspended before the rollover and resume after the rollover. Thus, the rollover has to be considered in the calculation of elapsed time.

When interrupts are disabled for periods of time longer than the clock tick duration, the milliseconds field may lose time. However, the seconds information remains accurate. Thus, in situations like these, compute elapsed time in seconds, not milliseconds.

Sample Programs: An example of the **DosGetDateTime** call is found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\WTCBAT\WTC.C
\TOOLKT12\MASM\SAMPLES\BSE\WTBAT\WT.ASM
```

Setting a Synchronous Timer

Issuing **DosSleep** with a milliseconds value of 0 causes the thread to relinquish the remainder of its current time slice to a ready thread of equal priority and to run when its next scheduled time slice comes up.

Specifying a value greater than 0 suspends the current thread for an interval that is equal to the value rounded up to the next clock tick.

When short intervals are requested, the factors of rounding up, thread priority interactions, and possible hardware interrupts may cause the wait period to be longer than is tolerable. In no case should DosSleep be used as a substitute for a real-time clock, because the rounding up of the wait interval will cause cumulative errors.

Sample Programs: Examples of the **DosSleep** call are found in the following subdirectories:

```
\TOOLKT12\MASM\SAMPLES\BSE\WTBAT\WT.ASM
```

Setting an Asynchronous One-Shot Timer

DosTimerAsync can be used to wait for a single event. Issuing this call with the handle of a system semaphore created with **DosCreateSem** allows a timer to run for a specified interval. The timer runs asynchronously while the thread of your application continues to execute. When the interval elapses, the system clears the semaphore.

To detect the expiration of the timer, the semaphore should be set with DosSemSet before issuing DosTimerAsync.

When desired, the timer can be stopped by issuing **DosTimerStop** with the timer handle returned by DosTimerAsync.

Setting an Asynchronous Recurring Timer

DosTimerStart can be used to perform an operation on a regular basis. This call causes an asynchronous timer to run and periodically clear a system semaphore.

To detect the expiration of a periodic interval, the semaphore should be set with **DosSemSet** before the next clearing of the semaphore occurs.

Sometimes, more than one clearing of the semaphore occurs before the thread resumes execution. When it is necessary to determine the actual expired time, issue DosGetInfoSeg and save the milliseconds field of the Global Information Segment before issuing DosTimerStart. Then compare this value with that of the milliseconds field once the thread is resumed. Remember that the range of accuracy can be within one or two clock ticks.

The semaphore is continually cleared until the timer is stopped. The timer can be stopped by issuing DosTimerStop with the timer handle returned by DosTimerStart.

Sample Programs: Examples of the **DosSemSet** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\MASM\SAMPLES\BSE\THREADB\THREAD.ASM
```

Dynamic Linking

Dynamic linking allows applications running in the system to make efficient use of memory by delaying the resolution of external references beyond LINK time to load time or run time. This feature enables an application to call a subroutine that is not part of its executable file, thus making the size of the executable file smaller. The subroutine called by the application can also be called by other applications. In addition, an application can decide whether or not to call a subroutine, depending upon events that occur at run time.

At the time an executable file is created with LINK, any or all of its segments can be defined in the module definition file as loadable before module execution begins (PRELOAD) or loadable only when referenced at run time (LOADONCALL). If this information is not specified for a segment, the segment is loaded only when referenced. LOADONCALL segments are identified in the .EXE or .DLL file header.

Load-time dynamic linking is the most frequently used method of dynamic linking. This is the way in which references to OS/2 API calls are made. OS/2 API routines are in segments defined as PRELOAD. References to these routines by an application are hard coded in the executable file created at LINK time. The IBM Linker/2 creates placeholders in the program module for their addresses. The references are then resolved by the dynamic link loader at load time.

Run-time dynamic linking allows an application to process externally supplied information (for example, a response from the user) before it loads a LOADONCALL dynamic link module created as a system extension.

Figure 34-4 shows the calling of a system extension through run-time dynamic linking. This figure depicts the program execution flow and the data handling aspects of providing a system extension as a callable routine. An extension may allocate data that is exclusive for each of its clients.

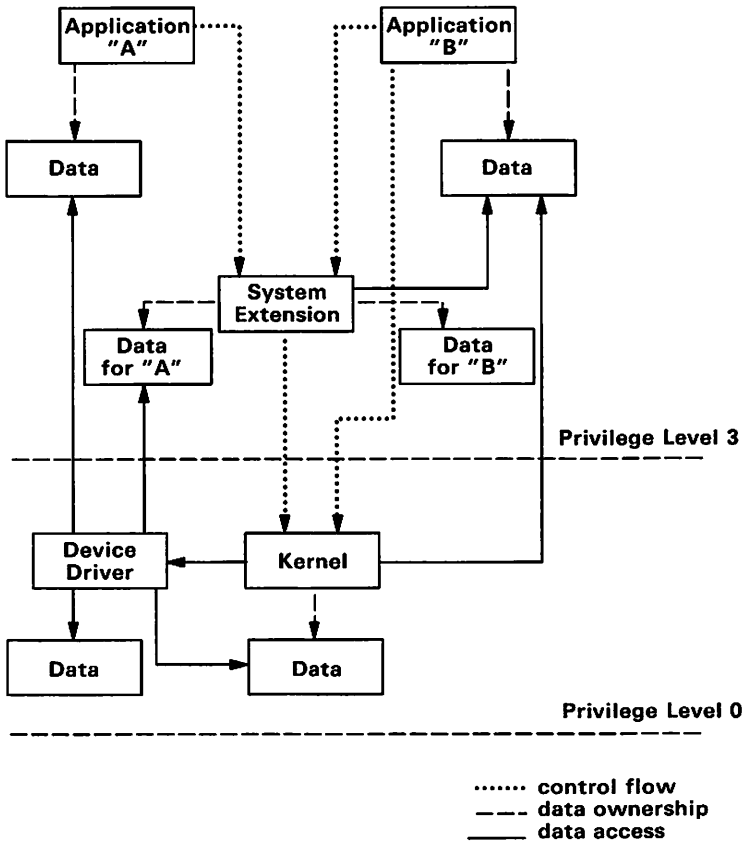


Figure 34-4. Loading a System Extension at Run Time

A good candidate for a system extension is a dynamic link module containing error processing routines. Because references to a dynamic link module containing LOADONCALL segments are not built into the application at LINK time, routines in the module can be added to or changed, and your application can retrieve these addresses at run time.

LOADONCALL segments, also referred to as load-on-demand segments, are handled by the loader similarly to segments that are swapped to disk. If swapping is enabled, the loader creates descriptors for the LOADONCALL segments in the LDT that indicate the segments are not present and reserves selectors for them. Control is then given to the module. When a LOADONCALL segment is referenced as a result of a run-time dynamic link call, a fault is generated, causing the loader to complete the segment's descriptor and load the segment from disk into memory.

If swapping has not been enabled, the loader loads all segments of a module, including any LOADONCALL segments, before it passes control to the module.

The DYNLINKC.C and DYNLINK.ASM sample programs call three different routines. Each of these routines demonstrates a different method for accessing a dynamic link module. The first module is preloaded by the system loader. The second module is loaded when called by the main program. The third module is explicitly loaded by the main program, called, and then freed.

Linking to a Dynamic Link Module at Run Time

A set of calls enable an application to load a dynamic link module at run time and reference its routines.

The following calls support run-time dynamic linking:

DosGetModHandle
: Determines if module already loaded and obtaining a handle for use with DosGetModName

DosGetModName
: Gets the full path name of a dynamic link module

DosLoadModule
: Loads a dynamic link module and obtaining a handle for use with DosGetProcAddr

DosGetProcAddr
: Gets the address to call a procedure

DosFreeModule
: Frees memory.

Loads a Dynamic Link Module

There are two classes of dynamic link modules that can be loaded at run time: global and specific. Global dynamic link modules can be shared by all processes in the system that reference the module name. A global module is located by the system by means of its path value in LIBPATH. Global dynamic link modules have unique names.

A specific dynamic link module is one created for an application's exclusive use. The specific dynamic link module is placed in a directory that has been defined by the application, and when specifying it with DosLoadModule, its full path name is given.

To determine the full path name of a module that is already loaded, issue **DosGetModHandle**. If the module is already loaded, DosGetModHandle returns a handle you can use with **DosGetModName**, which returns the drive, path, file name, and extension of the module. If the module is not loaded, an error code is returned by DosGetModHandle.

To load the module and attach the calling process to the module, issue **DosLoadModule**. If no path name characters are present in the name string, the module is assumed to reside in a file whose name is in the form, *ModuleName*.DLL. The file must be in a directory that has been identified in CONFIG.SYS with the LIBPATH command. Remember to specify the module name, not the file name.

Sample Programs: Examples of the **DosLoadModule** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\DLCBAT\DYNLINKC.C
\TOOLKT12\MASM\SAMPLES\BSE\DYNBLD\DYNLINK.ASM
```

Calling Dynamic Link Routines

To obtain the address of a procedure within a module, specify the module handle and the procedure name with **DosGetProcAddr**. The 32-bit address of the procedure, consisting of a selector and an offset, is returned. If the selector portion of the returned address is null, the offset portion is an explicit entry ordinal within the dynamic link module.

Note: The handle returned by DosGetModHandle should not be used with DosGetProcAddr for access to the already loaded dynamic link module. DosLoadModule must be issued to ensure the caller is attached to the module.

Freeing Modules

DosFreeModule frees the reference by a process to a dynamic link module. Each process that references the module must free it before the module can be discarded from system memory.

Sample Programs: Examples of the **DosFreeModule** and **DosGetProcAddr** calls are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\BSE\DLCBAT\DYNLINKC.C
\TOOLKT12\MASM\SAMPLES\BSE\DYNBLD\DYNLINK.ASM
```

Verifying Return Codes from Dynamic Link Calls

Each process running in the OS/2 multitasking environment requires a subset of the total resource available within the system, which is dependent on the user's configuration. In addition, dynamic link libraries can require system resource on a system-wide basis, on a process basis, or both. Therefore, applications should verify the return codes from dynamic link calls. The application should never assume that because the system allowed the program to load, there is sufficient system resource to successfully execute the application.

Debugging Aid to Program Authors

Because of the protected mode of operation of OS/2, special steps must be taken to allow a program such as a debugger to set breakpoints in a program under test. The following functions enable certain debug routine calls within OS/2:

DosPtrace
: Interface for program debugging

DosExecPgm
: Execute a program

DosStartSession
: Start an independent or child session.

Implementation of Breakpoint Debugging

To facilitate program debugging, OS/2 provides an interface into the kernel called **DosPtrace**. This call provides the means by which a process may control the execution of another process. It is intended for use in the implementation of breakpoint debugging using a debugger. The program under test, and the program being tested, must be executing in protected mode.

However, when the processor is running in protected mode, one process cannot directly manipulate the address space of another process. OS/2 makes this possible through the use of the trace option of **DosExecPgm** and the Ptrace buffer of DosPtrace.

Specifying a trace option of 2 with **DosStartSession** enables a debugger running in the parent session to trace all processes associated with an application running in a child session, regardless of whether a process has been started with DosStartSession or DosExecPgm. An existing queue is identified by the TermQ parameter, and the Related parameter is given a value of 1 on the DosStartSession call.

Starting with the session that is created with a trace option of 2, the session manager notifies the debugger by means of the queue whenever a session is created. The queue is posted, regardless of how the new session is started; independent or related, with or without inheritance. It is the responsibility of the debugger to resume execution of the new process with DosPtrace.

To receive notification of a session's creation, the debugger issues DosReadQueue with the NoWait parameter set to 0. The word returned in the request parameter of DosReadQueue has a value of 1. This will be the only process which will have addressability to the notification data element.

The format of the data element represented by the queue element is:

Size	Description
WORD	Session ID
WORD	Process ID.

After reading and processing the data element, the debugger must free the segment containing the data element by issuing DosFreeSeg.

The debugger can use DosSelectSession to switch itself, or any descendant session, to the foreground whenever one of its descendants is in the foreground.

When the kernel DosPtrace program receives control from the program being debugged, it returns to the debug program with the Ptrace buffer set to the current register contents and with indicators of the reason for return. When the debugger has control, the thread of the program that is being debugged will not be allowed to run.

The kernel DosPtrace program receives control upon a number of different events, including a breakpoint interrupt, some processor executions, and when the program ends.

Debugging a Process with Multiple Threads

DosPtrace supports debugging of a process with multiple threads by allowing the debugger to read and write registers, freeze and resume thread execution and get status on the threads.

To debug a process with multiple threads, set a field in the Ptrace buffer (Ptrace_B.TID) to the thread ID of the thread of interest. This causes the read and write register commands to receive only the register set of the specified thread.

The address space for a process with multiple threads is the same for all the threads in the process. However, when issuing commands to read and write memory locations, or set breakpoints, all the threads in the process are affected even though the command was issued for a specific thread.

Through the use of the TRC_C_Freeze and TRC_C_Resume commands, the debugger may suspend and resume specific threads. This is useful in having only those selected threads be affected by the breakpoints and manipulating them while other threads remain suspended.

Terminating the Debug Process

When a debugger process terminates, the program being debugged also terminates. To do this, an internal link between the debugger and the program being debugged is maintained. This link is established upon the first successful DosPtrace command. Once established, the link cannot be reset.

The program being debugged does not need to be a direct child. As a result of this, there is a small window of time between the DosExecPgm call and the first DosPtrace call, where if the debugger terminates, the program being debugged cannot be cleaned up. If this the case, the system terminates the program being debugged in a few minutes.

Chapter 35. Communications Between Threads and Processes

This chapter describes the mechanisms that enable you to coordinate the execution of threads and pass data between processes.

The OS/2 tools package does not provide sample programs for all function calls discussed in this chapter. However, when an example of a function call is used in a sample program, the subdirectory path of the sample program is shown. **Note that a sample program may not illustrate the task of the function call as discussed in the chapter text.**

Creating and Accessing Semaphores

A semaphore is an object that is manipulated by threads issuing semaphore function calls. The semaphore can be used to serialize access to a resource. It can also be used to signal the occurrence of an event to waiting threads.

To gain access to a resource that must be serially used, the thread requests a semaphore. Once its request is successful, the thread uses the resource, and any requests for the semaphore by other threads are blocked. When the thread is through using the resource, it clears the semaphore. Any threads blocked on requests for the semaphore resume executing so that one of them can claim the semaphore and use the resource.

To wait for an event to occur, a thread sets the semaphore for signaling purposes and then waits for notification that the event has occurred. The clearing of the semaphore by another thread signals the event's occurrence (such as the placing of data in a buffer) and causes the waiting thread to resume its execution.

The semaphore calls that enable threads to perform these functions are discussed in detail under "Serializing the Use of Resources" on page 35-3 and "Waiting for the Occurrence of an Event" on page 35-4.

OS/2 recognizes two types of semaphores:

- System semaphores
- RAM semaphores.

A system semaphore is a named object defined by OS/2, which resides in a memory buffer and is defined within the file-system name space as a pseudo-file. Processes obtain ownership of a semaphore with a successful **DosSemRequest** call.

Sample Programs: An example of the **DosSemRequest** call is found in the following subdirectory:

```
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
```

A RAM semaphore is a doubleword of storage defined by your application. If the RAM semaphore is defined as an *FS RAM* semaphore, it has characteristics in common with a system semaphore.

The following table shows the common uses of OS/2 semaphores.

Use	Semaphore
Serialization of threads of the same process	RAM
Serialization of threads of different processes	FS RAM
Signaling of threads of the same process	RAM
Signaling of threads of different processes	System, Nonexclusive

FS RAM semaphores and exclusive system semaphores are well suited for serializing the use of resources by threads of different processes. They both track semaphore ownership, which means that if the process of the semaphore owner ends, the resource can be tidied up by an exit list routine.

The concept of semaphore ownership is important to understand. A semaphore is considered *owned* by the thread that currently holds the semaphore. OS/2 keeps track of thread and process IDs associated with the owned semaphore.

FS RAM and system semaphores support recursive routines. A thread that owns the semaphore because it has issued DosFSRamSemRequest or DosSemRequest can issue a request for the

semaphore again without first clearing the semaphore. If this is done with a RAM semaphore, a deadlock ensues because the owning thread blocks on the second request for the semaphore, waiting for itself to clear the semaphore.

System Semaphores

A system semaphore is created by OS/2 when a thread issues **DosCreateSem**. A name must be specified with DosCreateSem that includes the prefix \SEM\. DosCreateSem initializes the semaphore as unowned and returns a handle, which is used to access the semaphore.

At the time of its creation, a system semaphore can be specified as being for the *exclusive* use of the owning process. For example, if a system semaphore is created as exclusive, only other threads within the process can modify the semaphore.

Alternatively, the system semaphore can be created as *nonexclusive*. A nonexclusive system semaphore can be modified by any process in the system that knows its name.

Both the exclusive and the nonexclusive system semaphore have a use count, which enables them to support recursive routines. The use count records how many times the owning thread has issued DosSemRequest without a corresponding DosSemClear. The semaphore is not released until the use count is 0.

If a process ends during the time its thread has claimed the system semaphore but has not yet released it, OS/2 can release the semaphore and notify any waiting threads that the owner of the semaphore has ended while holding the semaphore.

Accessing a System Semaphore Created by Another Process

To access a system semaphore created by another process, issue **DosOpenSem** and specify the pointer to the ASCIIZ string name of the semaphore. The open call does not test or change the state of the semaphore.

When a process with open semaphore handles issues a DosExecPgm, the child process inherits any open semaphore handles. All inherited semaphores are initially not owned by the child process, even though the parent process may have owned them at the time the DosExecPgm was issued. This is because only one process can own a semaphore at a time.

Closing the System Semaphore

To close the system semaphore, issue **DosCloseSem**. OS/2 system semaphores reside in a memory buffer, rather than on a disk file. This means that when all the processes that have opened a semaphore issue DosCloseSem, the system semaphore is deleted and must be re-created by the next user of the semaphore.

RAM Semaphores

A RAM semaphore is a doubleword of storage that is defined in your application. The doubleword initialized to 0 indicates the semaphore is unowned and can be claimed by a thread issuing a DosSemRequest.

A RAM semaphore provides a performance advantage over a system semaphore, because the RAM semaphore handle used with semaphore function calls is its actual address.

Processes that use the RAM semaphore for signaling purposes must agree on the definition of the semaphore and have shared access to the area of memory where the semaphore is defined.

RAM semaphores accessed with DosSemRequest and DosSemClear calls do not support recursive routines. In addition, there is no capability for tracking RAM semaphore ownership. If a process ends while owning the RAM semaphore, cleanup operations cannot be performed to protect the integrity of the resource being used.

Although the RAM semaphore should not be used to serialize access to a resource by threads of different processes, it is safe to use for serialization by threads of the same process.

Defining a RAM Semaphore as a Fast-Safe RAM Semaphore

OS/2 allows you to define a RAM semaphore with a Fast-Safe (FS) RAM data structure, which provides fields for tracking semaphore ownership. Thus, the FS RAM semaphore combines the performance characteristics of a RAM semaphore with the safety characteristics of a system semaphore for

serialization of access to a resource by threads of different processes.

To claim and release an FS RAM semaphore, a thread issues **DosFSRamSemRequest** and **DosFSRamSemClear** calls. These are the only calls an FS RAM semaphore may use. The format of the Fast-Safe RAM data structure is shown in Figure 35-1. Before the initial DosFsRamSemRequest call, the entire structure must be initialized to 0, and the length must be set to 12.

Like the system semaphore, an FS RAM semaphore has a use count associated with it. Recursive requests for FS RAM semaphores are supported by the fs_UseCount field, which is incremented and decremented by DosFSRamSemRequest and DosFSRamSemClear calls. Ownership of the FS RAM semaphore is not given up until it has been cleared the same number of times it is requested.

If a process ends while owning a Fast-Safe RAM semaphore, an associated exit list routine can issue a DosFSRamSemRequest to override the owning thread ID and perform normal cleanup operations. Upon completion, an exit list routine restores the resource for use by issuing DosFSRamSemClear. The fs_Client field, which is initialized to 0 when the semaphore is acquired, allows the owning process to describe the resource currently being accessed. This information is useful to the exit list routine. For more information on exit list routines, see "Performing Cleanup Operations When a Process Ends" on page 34-9.

Other than the fs_Client field, the caller should not modify any fields in the data structure. An exit list routine may access, but not modify, both the fs_ProcID field to determine if it owns the semaphore, and the fs_UseCount field to determine how many times to release the semaphore.

Comparing System, Ram, and Fast-Safe Ram Semaphores

Ram semaphores are the fastest semaphores. However, the following unfavorable conditions exist:

- The application must provide memory for the semaphore. This memory must be shared between all processes which use the semaphore.
- The semaphore is not cleared if the owner dies. This can cause the processes waiting on the semaphore to deadlock.
- Since the semaphore is kept in application memory, the application can inadvertently alter the semaphore value.

Fast-safe ram semaphores are slower than ram semaphores, but if the owner dies the semaphore will be cleared. However, all other unfavorable conditions which exist for ram semaphores apply to fast-safe ram semaphores.

System semaphores are the slowest semaphores and have no unfavorable conditions. For example, system semaphores are kept in system memory and altered only by OS/2. Also, system semaphores are cleared if the owner dies.

```
FSRamSemStructure   struc
    fs_Length   dw   14   ; Length of this structure (in bytes)
    fs_ProcID   dw   0    ; Process ID of owner, or zero
    fs_ThrdID   dw   0    ; Thread ID of owner, or zero
    fs_UseCount dw   0    ; Reference count
    fs_Client   dw   0    ; 16-bit field for use by owner
    fs_RamSem   dw   0    ; OS/2 RAM semaphore
FSRamSemStructure   ends
```

Figure 35-1. Fast-Safe RAM Data Structure

Serializing the Use of Resources

The following function calls support controlling access to a serially reusable resource:

DosSemRequest
: Claiming or checking the status of a system semaphore or a RAM semaphore

DosSemClear
: Releasing a system semaphore or a RAM semaphore

DosFSRamSemRequest
: Claiming or checking the status of an FS RAM semaphore

DosFSRamSemClear
: Releasing an FS RAM semaphore.

Figure 35-2 on page 35-5 contains a pseudocode example that demonstrates the use of DosSemRequest and DosSemClear calls with a RAM semaphore to control access by threads of the same process to a resource. The application intermixes data from two separate resources into a single buffer and terminates if an error occurs writing to the buffer.

Accessing the Resource

Access to the resource is obtained when ownership of the semaphore is established by a successful **DosSemRequest** or **DosFSRamSemRequest**. When another thread attempts to access the resource by requesting the semaphore with DosSemRequest or DosFSRamSemRequest, the requesting thread is blocked. If a timeout value is associated with the request, the request is tried again after the timeout occurs.

If a thread is terminated while owning a system semaphore, ERROR_SEM_OWNER_DIED is returned to the thread that gets the semaphore next because of a DosSemRequest. The thread can release the resource by issuing a DosSemClear.

If a process is terminated while owning an FS RAM semaphore, the process' exit list routine is given control (if one exists). The routine can examine all its FS RAM semaphore data structures for any that are owned by the process (as indicated by fs_ProcID) and take the appropriate steps to protect the resource.

Freeing the Resource

When a thread has finished using the protected resource, it makes it available for the next thread to use by releasing the semaphore with **DosSemClear** or **DosFSRamSemClear**.

Recursive requests for both exclusive system semaphores and FS RAM semaphores are supported by means of use counts, which keep track of the number of times the semaphore is requested without a corresponding clearing call. When the use count is 0, the clearing call resets the semaphore as unowned.

Any threads that are blocked on the semaphore resume execution so the next thread can claim the semaphore with DosSemRequest or DosFSRamSemRequest. A thread that is blocked on either of these calls does not resume executing until it successfully claims the semaphore.

Sample Programs: Examples of the **DosSemClear** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\MASM\SAMPLES\BSE\THREADB\THREAD.ASM
```

Waiting for the Occurrence of an Event

The following function calls support waiting for an event and signaling its occurrence:

DosSemSet
: Setting the semaphore until an event occurs

DosSemWait
: Waiting for a specified amount of time or until the semaphore is cleared

DosSemSetWait
: Setting and waiting on a semaphore or returning immediately

DosMuxSemWait
: Waiting until one of multiple semaphores is cleared

DosSemClear
: Clearing the semaphore to signal that event has occurred.

Setting and Waiting for a Semaphore

A thread that needs to wait for the occurrence of an event sets the semaphore for signaling purposes by issuing **DosSemSet**. If the thread is going to be signaled by a thread in its own process, it sets a RAM semaphore. If the thread is going to be signaled by a thread of another process, either a nonexclusive or a RAM semaphore can be used. If a RAM semaphore is used, the processes must have shared access to the area in memory where the RAM semaphore is defined. DosSemSet sets the semaphore regardless of its current state.

Sample Programs: Examples of the **DosSemSet** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC1\GRAPHIC1.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\MASM\SAMPLES\BSE\THREADB\THREAD.ASM
```

After setting the semaphore, the thread then issues **DosSemWait** to wait to be signaled that the event has occurred. This call blocks the thread until the semaphore is cleared. Once the thread is cleared by the signaling thread with a DosSemClear, the thread that is blocked resumes execution.

Main Routine

CALL DosAllocSeg to allocate storage for application's resources:

Allocate and initialize Buffer X, which will receive data from resources C and D.

Allocate and initialize RAM Semaphore Y, which will be used to control access to Buffer X.

Allocate and initialize RAM Semaphore Z, which will be used by threads A and B to signal the MAIN thread that error has occurred in writing to Buffer X.

CALL DosSemSet to set RAM Semaphore Z.

CALL DosCreateThread to create thread A, which executes the A Routine.

CALL DosCreateThread to create thread B, which executes the B Routine.

CALL DosSemWait to wait for RAM Semaphore Z to be cleared by thread A or B.

CALL DosExit to terminate the application.

A Routine

UNTIL an error occurs

Get data from Resource C.

CALL DosSemRequest to request RAM Semaphore Y. Block if thread B has claimed the semaphore. When this call returns, thread B cannot use the semaphore until it is cleared.

Write data to Buffer X.

CALL DosSemClear to clear RAM Semaphore Y so that other thread can use the semaphore and the buffer resource it protects.

END UNTIL

CALL DosSemClear to clear RAM semaphore Z and signal the MAIN thread to terminate application because error has occurred.

B Routine

UNTIL an error occurs

Get data from resource D.

CALL DosSemRequest to request RAM Semaphore Y. Block if thread A has claimed the semaphore. When this call returns, thread A cannot use the semaphore until it is cleared.

Write data to Buffer X.

CALL DosSemClear to clear RAM Semaphore Y so that other thread can use the semaphore and the buffer resource it protects.

END UNTIL

CALL DosSemClear to clear RAM semaphore Z and signal the MAIN thread to terminate application because error has occurred.

Figure 35-2. Synchronizing Two Threads within a Process

Sample Programs: Examples of the **DosSemWait** call are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\GRAPHIC1\GRAPHIC1.C
\TOOLKT12\C\SAMPLES\PM\GRAPHIC2\GRAPHIC2.C
\TOOLKT12\MASM\SAMPLES\BSE\THREADB\THREAD.ASM
```

Figure 35-2 on page 35-5 illustrates one way in which semaphore calls can be used to signal the occurrence of an event. In this example, a RAM semaphore is used by threads of the same process to signal the occurrence of an error.

In some situations, there can be a problem ensuring that the semaphore remains set when DosSemSet and DosSemWait calls are used. During the time between the issuing of these two calls, the semaphore can be cleared. Issuing DosSemWait on a semaphore that has been cleared has no effect. Instead of blocking, the caller continues to execute.

DosSemSetWait combines the operations of verifying that the semaphore is set and waiting for the set semaphore. This call blocks the current thread until the next DosSemClear is issued and the thread determines the semaphore is clear.

Waiting for One of Multiple Semaphores to Clear

A process that needs to wait for more than one event to occur can issue **DosMuxSemWait**. This allows a process to wait until one of a list of semaphores is cleared. A process can wait for up to 16 semaphores at once.

If all the semaphores are set, the thread is blocked. If a semaphore clears, the thread returns from DosMuxSemWait.

DosMuxSemWait is known as an edge-triggered procedure. It is possible for the semaphore that was cleared to be reset by the time the thread returns from DosMuxSemWait.

One of the events a process can wait for is the adding of an element to a queue. See "Waiting for an Element in a Queue" on page 35-10.

Setting Flags to Respond to Time-Critical Events

OS/2 has a signal mechanism that generates signals when certain events occur. An application can accept the default action taken by the system when a signal is generated, or it can respond to the signal if it has registered a signal handler.

The following function calls support managing signals generated by the OS/2 signal mechanism:

DosSetSigHandler
: Identifying a signal handler to the system

DosFlagProcess
: Notifying a process that an event defined by the application has occurred.

DosHoldSignal
: Disabling, enabling signal processing

DosSendSignal
: Sending a Ctrl+C or Ctrl+Break signal.

Signals are generated by two types of events: trap events and external events.

Trap events are synchronous, having been caused by an instruction executed by the thread of a process. Most trap events (such as divide by 0) are handled by means of the **DosSetVec** function call. All other events that generate a signal are asynchronous external events.

Registering a signal handler by issuing **DosSetSigHandler** allows a process to respond to signals generated by specific trap and external events. Events for which signal handlers can be registered are:

- Ctrl+Break (or Ctrl+C) keys pressed (SIGBREAK, SIGINTR)
- Process terminated by DosKillProcess (SIGTERM)
- Setting of Process Flag A (SIGPFA)
- Setting of Process Flag B (SIGPFB)
- Setting of Process Flag C (SIGPFC).

DosFlagProcess sets one of three user-defined flags, which causes OS/2 to send a signal to the process whose process ID is specified by DosFlagProcess. To respond to the signal, the process registers a signal handler with DosSetSigHandler for the signal number corresponding to the user flag. By default, the signal generated by the setting of a user flag is ignored by a process. A process may also specify that the flag action be ignored and that an error

code is to be returned to the flagger. The error code *process is zombie* indicates that the flagged process died, but the DosCWait function allows the process to issue the return code.

If the process has specified a signal handling address and the exception is a trap event, the thread initiating the trap executes the signal handler routine.

If the process has specified a signal handling address and the exception is an external event, thread 1 (the initial thread of execution started by OS/2) is diverted, in a forced far call similar to a hardware interrupt, to the proper signal handler address. If thread 1 is in the midst of a function call that does not complete quickly, the function call is stopped because a signal represents a time-critical event. The signal interrupt takes place immediately upon return from the OS/2 function call.

An application should reserve its initial thread of execution to handle signals and use another thread for its normal operations. Thread 1 can be reserved by having it block on an eternally reserved RAM Semaphore.

DosSendSignal sends either a Ctrl+C (SIGINTR) or a Ctrl+Break (SIGBREAK) signal to the last process in a command subtree that has a corresponding signal handler installed. A command subtree is all of the processes created as a result of a single command. The selected process may have descendents, but none of them will have the handler installed for the signal. Note that only Vio-windowed and full-screen applications support Ctrl+C and Ctrl+Break signal handles.

Communicating with Pipes and Queues

In addition to using semaphores and shared memory, processes can communicate with one another using these mechanisms:

- Unnamed pipes
- Queues
- Named pipes.

Unnamed pipes allow related processes to communicate as if they were doing file I/O. For example, if a program that lists a file were to specify its input or output as a pipe instead of a file, the program could operate unchanged.

Queues provide a means of communication between unrelated processes and offer certain advantages over unnamed pipes. Because only a pointer to data is passed, not the data itself, a queue can be a more efficient means of communication. In addition, the total size of messages represented in the queue is not bound, as a pipe is, by a 64KB segment.

Named pipes provide the most powerful method of communication between unrelated processes. Without named pipe I/O, the only way unrelated processes can communicate with one another is through shared memory, or through queues that depend on passing references to shared memory. Sharing of memory between unrelated processes may be impractical to achieve.

Processes using named pipe function calls can communicate with one another locally, or they can have their requests redirected across a network, such as a Local Area Network (LAN).

The needs of your application should determine which of these mechanisms to use.

Passing Data in an Unnamed Pipe

Using read and write calls, closely related processes can store and retrieve data in an unnamed pipe. An unnamed pipe consists of a circular buffer created in memory, whose maximum size is 64KB.

A child process created with **DosExecPgm** inherits handles to unnamed pipes from its parent process. Thus, the parent process can pass a pipe handle for the child process to use by means of shared memory.

Sample Programs: An example of the **DosExecPgm** call is found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Another way communication between related processes using unnamed pipes can be achieved is to duplicate pipe handles as standard I/O handles. This is done with the **DosDupHandle** call. See "Creating a Duplicate Handle" on page 33-14.

The following function calls support using unnamed pipes:

DosMakePipe
Creating the pipe
DosRead, DosReadAsync
Reading data from the pipe
DosWrite, DosWriteAsync
Writing data to the pipe
DosClose
Closing the pipe.

Creating an Unnamed Pipe

To create an unnamed pipe, issue **DosMakePipe** and specify the pipe's size. The system returns a read handle and a write handle, which are used to access the pipe.

Sample Programs: An example of the **DosMakePipe** call is found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Adding Data to the Unnamed Pipe

To add data to the unnamed pipe, issue **DosWrite** or **DosWriteAsync** and specify the pipe's write handle.

Write requests to a pipe are processed in the order they are made. As each request is processed, subsequent requests are blocked until the current request finishes writing to the pipe.

Multiple writes can be made before a read occurs. When the pipe becomes full, write requests are blocked until space is freed by read requests.

Sample Programs: An example of the **DosWrite** call is found in the following subdirectory:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
```

Removing Data from the Unnamed Pipe

To remove data from the pipe, issue **DosRead** or **DosReadAsync** and specify the pipe's read handle.

Subsequent reads before another write can empty the pipe.

Sample Programs: An example of the **DosReadAsync** call can be found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROG.2
```

Deleting an Unnamed Pipe from Memory

Issuing **DosClose** enables an unnamed pipe to be deleted from memory. The pipe is not deleted until all requesting processes close the pipe.

Passing Addresses to Data with a Queue

A queue is a named object that is used to pass packets of information between threads of the same process or different processes. The packets, called queue elements, contain the addresses and lengths of data areas in shared memory segments. These segments are accessible to the reader of the queue.

Queuing calls enable a process to create and own a queue, which can be written to by other processes, called client processes. The client processes can add elements to the queue, and the owning process can remove them.

Ownership and Management of a Queue

The creator and owner of a queue has queue-managing privileges. Only the owning process and its threads may:

- Peek at queue elements without removing them
- Remove elements from the queue
- Purge the queue of all its elements
- Write to the queue without opening it
- Delete the queue from memory.

The following function calls support queue activities performed by the owning process and its threads:

DosCreateQueue
Creating the queue
DosPeekQueue
Examining a queue element without removing it
DosReadQueue
Removing an element from the queue
DosPurgeQueue
Purging all the elements in a queue
DosQueryQueue
Querying the number of elements in a queue
DosWriteQueue
Adding an element to the queue (by another thread in the same process)

DosCloseQueue
Deleting a queue from memory.

Client Access to a Queue

The following function calls support client access to a queue:

DosQueryQueue
Querying the number of elements in a queue
DosOpenQueue
Accessing the queue to add an element
DosWriteQueue
Adding an element to the queue
DosCloseQueue
Terminating access to the queue.

Creating a Queue

The creator and owner of the queue determines whether the ordering of queue elements is based on arrival or priority. If the ordering is based on priority, priority values for the data are agreed upon by processes using the queue.

To create a queue, issue **DosCreateQueue** and specify the name of the queue and the data ordering type (FIFO, LIFO, or Priority). The ASCIIZ name must include a \QUEUES\ prefix. This name is also specified when a client process opens the queue with DosOpenQueue. The system returns a queue handle.

Sample Programs: Examples of the **DosCreateQueue** call are found in the following subdirectories:

```
\TOOLKT12\MASM\SAMPLES/BSE\PROCESSB\PROG1.ASM
```

Adding an Element to the Queue

When a thread of the owning process needs to add an element to the queue, it merely issues DosWriteQueue. However, before another process can add an element to the queue, it has to open the queue with **DosOpenQueue**.

To add an element to the queue, issue **DosWriteQueue**. If the queue is a priority-based queue, specify the priority of the element being written.

If the owning process has terminated or has closed the queue before DosWriteQueue is issued, QUE_INVALID_HANDLE is returned.

Sample Programs: An example of the **DosOpenQueue** and the **DosWriteQueue** call is found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Peeking at a Queue Element

Only the queue owner can issue **DosPeekQueue**. This call allows the owning process to examine a queue element without removing it from the queue. A DosPeekQueue is similar to a non-destructive read.

If the queue is empty and a wait is specified, the requesting thread waits until an element is placed in the queue. If the queue is empty and NoWait is set, DosPeekQueue returns with ERROR_QUE_EMPTY.

The element code parameter can be used by successive peek requests. When the first DosPeekQueue request is issued, the owning process sets this field to 0, indicating the peek is to start with the first element in the queue. When DosPeekQueue returns, this field contains an identifier for the queue element that has been looked at. A subsequent peek with this identifier specified returns the next element with its corresponding identifier. These identifiers can be used in conjunction with DosReadQueue requests.

Removing an Element from the Queue

The owning process issues **DosReadQueue** to remove an element from the queue. If the queue is empty and a wait is specified, the requesting thread waits until an element is placed in the queue. If the queue is empty and NoWait is set, DosReadQueue returns with ERROR_QUE_EMPTY.

Like DosPeekQueue, DosReadQueue has an element code parameter, which overrides the normal FIFO or LIFO read ordering that was specified with DosCreateQueue. This parameter allows a thread to specify an identifier with DosReadQueue that was returned by a DosPeekQueue, so it can read an element examined by DosPeekQueue. The element read can then be compared with element returned by DosPeekQueue.

Sample Programs: An example of the **DosReadQueue** call is found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROG1.ASM
```

Waiting for an Element in a Queue

If the Wait option is selected for either **DosPeekQueue** or **DosReadQueue**, the handle of a system or RAM semaphore can be specified, which is to be cleared when the queue has data placed in it. This semaphore is typically used by **DosMuxSemWait** to wait on a queue or any of several other events.

If the handle is for a RAM semaphore, the semaphore must be in a segment that is shared by the queue owner's process and any client process that issues a DosWriteQueue.

If the owning process clears a semaphore when it issues DosReadQueue or DosPeekQueue, client processes must first issue DosOpenSem for the system semaphore.

If multiple threads of the owning process are processing elements from the queue, the same semaphore must be provided on all DosPeekQueue and DosReadQueue requests. (This handle is ignored if NoWait is set to 0.)

For more information on the DosMuxSemWait call, see "Waiting for One of Multiple Semaphores to Clear" on page 35-6.

Querying the Number of Elements in a Queue

The owning process and processes that write to the queue can issue **DosQueryQueue** to determine the number of elements that are currently in the queue.

If the owning process closes the queue before this request is made, ERROR_QUE_INVALID_HANDLE is returned.

Emptying the Queue

The owning process issues **DosPurgeQueue** to empty the queue. This is an unconditional purge.

Terminating Access to a Queue and Deleting It from Memory

The processes that write to the queue issue **DosCloseQueue** to terminate their access to the queue. However, the queue is not deleted from memory until the owning process issues the close call. If the owner has closed the queue, a process that attempts to write to the queue receives ERROR_QUE_INVALID_HANDLE.

Sample Programs: Examples of the **DosCloseQueue** call are found in the following subdirectories:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

Managing Queue Data Storage

The PROCESS.ASM and PROG1.ASM sample programs illustrate the passing of data between a parent process and a child process, using a queue and shared memory. See "Creating and Accessing Shared Segments" on page 32-3 for descriptions of how to allocate shared segments.

The sections that follow describe several methods for managing the shared memory needed to hold the data. The method you select depends on the requirements of your application.

A Named Shared Segment

As a solution for processes that are cooperating closely and communicating with small messages, or at low data rates, manage the data element storage in a single shared segment.

The name of the shared segment is established by agreement among the parties involved. For simplicity, the name can be the same as the queue name.

A limit is imposed by the single 64KB segment size available for data elements. If the server process is available for long periods of time, or if only short messages are being sent, this should not be a problem. However, the 64KB limit may not be acceptable if long messages are being sent or if the server process is unable to maintain requests from several clients. The 64KB limit applies to the sum of all messages of all client processes.

A Queue-Driven Server

For a more complex server-client implementation, where many potential client processes are driving a server, use a queue-driven server that provides support for several processes.

For this example, assume the server provides either one queue for all clients or a separate queue for each client.

A table is kept, with each entry comprising:

- The Process ID (PID)
- Shared memory segment name
- Shared memory handle.

On any call, the server needs only to get the currently active PID and find the entry in the table that matches the PID :

- If the PID is found, the server gets the memory handle to be used for this process.
- If the PID is not found, this must be the first request by the process. The server creates the shared segment and sets up a table entry for this process.

When the server close function is received, the server must remove this process entry from the table.

Figure 35-3 on page 35-12 shows a queue passing data to a server process Y from three child processes D, E, and F.

Process Y can examine each element in the queue and remove only those that it needs.

A Locally Shared Segment

It is also possible to allocate a data segment with DosAllocSeg and use DosGiveSeg to pass addressability to the server. This method allows a message to vary its size up to the 64KB limit.

For this technique, the following steps can be used:

- Save the process ID of the queue owner when DosOpenQueue is issued.
- Allocate the queue element segment by issuing DosAllocSeg, specifying the segment to be shared.
- Pass addressability to the segment from the client to the server, using DosGiveSeg.
- Issue DosWriteQueue to reference a segment and provide the RecipientSegSelector returned from DosGiveSeg in the Data Buffer field.

After the client completes its queue requests or terminates, the server issues DosFreeSeg to release the segment.

Deleting Queue Elements

Some server applications may require the ability to delete individual elements from a queue. For example, when a client process terminates, it may be necessary to delete the remaining elements from that process.

This can be accomplished by setting up a loop similar to the following:

```
Do until end_of_queue
  DosPeekQueue NoWait
  If have element AND its PID of interest
    Then
      DosReadQueue element peeked above
Enddo
```

Two-Way Communication with Named Pipes

Named pipes provide a two-way means of communication between unrelated processes. Processes can communicate either locally, or remotely across a network. If communication is across a network, named pipe requests are redirected by the system.

Ownership and Management of the Named Pipe

Named pipes provide two-way communication between a server process and a number of client processes. The server creates the pipe and then waits for a client to open the pipe at the other end to make a request. The client makes the request, and the server responds to it.

The process at each end of the pipe has its own handle. The server handle is returned with DosMakeNmPipe, and the client handle is returned by DosOpen. Unlike unnamed pipes, each end can use its handle to both read and write. Because there are two pipe buffers for a named pipe (one for each direction of communication), anything that one end writes, the other can read. Thus, the pipe acts like a duplex channel between the processes.

The following function calls support activities performed by the named pipe server process:

DosMakeNmPipe
: Creating the named pipe

DosConnectNmPipe
: Waiting for a client to access the pipe

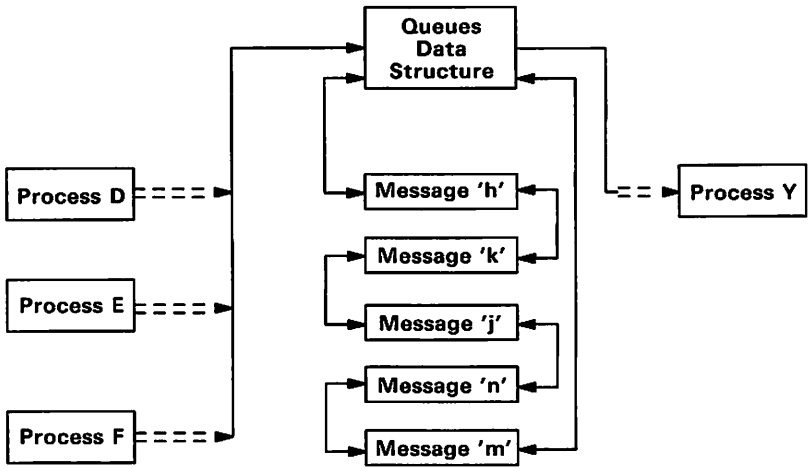


Figure 35-3. Passing Queue Data from Three Child Processes

DosRead/DosReadAsync
Reading a request from the client
DosWrite/DosWriteAsync
Writing a response to the client
DosDisConnectNmPipe
Preparing for the next client of the pipe
DosClose
Deallocating the pipe.

Creating a Named Pipe

Issuing **DosMakeNmPipe** creates the named pipe. A named pipe has as its name an ASCIIZ string in the form *d*:\PIPE*filename*.

Setting the file write-through bit has significance only for network communication. This allows data to be sent across the network as soon as it is written.

An inheritance flag indicates whether the pipe handle is inherited by a child process or is private to the creating process.

The access mode of the pipe can be set for inbound (client to server), outbound (server to client), or full duplex (two-way) communication.

The blocking bit can be set so reads or writes block if no data is available or return immediately.

A named pipe can be created as a message pipe or a byte pipe. If the pipe is created as a message pipe, each write is a message, whose size is the length of the write. The first two bytes of each message, called the header, contain the message length. The message length amount *does not* include the two bytes of the header. Zero length messages are not allowed. DosWrite automatically encodes message lengths in a message pipe, so applications need not encode this information in the buffer being written.

The read mode of a message pipe can be set to read pipe data as either a byte stream or a message stream. When a message pipe is read in byte stream mode, DosRead skips over the message headers.

Pipes created as byte pipes are read only as byte streams. No headers are created for byte stream pipes.

A named pipe can have multiple instances created by multiple server processes. The first instance of a named pipe is created when its name is first specified with DosMakeNmPipe. At this time, the named pipe is defined to the system, and an instance count is specified, which determines how many instances of the named pipe can be created. Depending on the value specified, the pipe can be unique, or it can have a specified or unlimited number of instances. Each pipe instance created by a DosMakeNmPipe call has a unique handle and can be accessed by only one client at a time.

The advantage of multiple instances is that the named pipe does not have to be redefined to the system, and multiple clients can access multiple instances of the named pipe at the same time.

At the time of the pipe's creation, the server process specifies the sizes of the in-buffer and out-buffer. If the pipe is a message pipe and the size of the messages to be received is known, the server can

control how many messages at a time can be placed in the buffer.

Waiting for a Client to Open the Pipe

Issuing **DosConnectNmPipe** by the server process enables the server to wait for a client to open the pipe. The newly made or recently disconnected pipe enters a listening state that allows acceptance of a DosOpen issued by a client process.

If the client end of the pipe is open at the time DosConnectNmPipe is issued, DosConnectNmPipe returns immediately and has no effect. If the client end is not open, DosConnectNmPipe either waits until it is opened (when blocking mode is set) or returns immediately with ERROR_PIPE_NOT_CONNECTED (when non-blocking mode is set). In the case where ERROR_PIPE_NOT_CONNECTED is returned, the pipe enters a listening state, permitting a client to issue a successful DosOpen.

If the pipe has been opened and closed by the client but is not yet disconnected by the server, DosConnectNmPipe always returns ERROR_PIPE_BROKEN.

Multiple DosConnectNmPipe calls can be issued in nonblocking mode. If the pipe is not opening or closing, the first DosConnectNmPipe puts the pipe in a listening state, and subsequent ones simply test the pipe state.

If DosConnectNmPipe is called by a client process, ERROR_BAD_PIPE is returned.

If the waiting by a DosConnectNmPipe call for the client to open the pipe is interrupted, ERROR_INTERRUPT is returned.

Preparing the Pipe for the Next Client

The server issues **DosDisConnectNmPipe** followed by DosConnectNmPipe to prepare the pipe for the next client.

If the client end of the pipe is currently open when DosDisConnectNmPipe is issued, it is forced closed, and the client gets an error code on its next operation. In such a case, data may be discarded that has not been read by the client. A client that gets forced off the pipe must still issue DosClose, because a DosDisConnectNmPipe invalidates the pipe's handle but does not free it.

The serving end may also issue a DosClose without a preceding DosDisConnectNmPipe to free a pipe handle. This enables the client to read any data remaining in the buffer.

If the client end is currently closing, DosDisConnectNmPipe acknowledges the close, and DosConnectNmPipe makes the pipe available for reopening by the next client.

As a result of DosDisConnectNmPipe, any threads blocked on the pipe are awakened. Threads blocked on the pipe because of DosWrite receive ERROR_PIPE_BROKEN. Threads blocked on the pipe because of DosRead receive a BytesRead = 0, indicating end-of-file.

Deallocating the Pipe

A pipe is considered broken when all the handles accessing one end have been closed. If the server process issues **DosClose** and the pipe is already broken, it is deallocated immediately. Otherwise, the pipe is deallocated when the last client handle is closed.

Accessing the Named Pipe by a Client

The following function calls support activities performed by a named pipe client process:

DosOpen
: Gaining access to the named pipe

DosWaitNmPipe
: Waiting for an available pipe instance

DosWrite/DosWriteAsync
: Writing a request to the server

DosRead/DosReadAsync
: Reading a response from the server

DosClose
: Ending use of the pipe.

Gaining Access to the Named Pipe

Issuing **DosOpen** opens the client end of the pipe and returns a handle. The pipe must be in a listening state brought about by the server process issuing DosConnectNmPipe; otherwise, the open fails and ERROR_PIPE_BUSY is returned.

When a named pipe is opened, its initial state is set to block reads and writes, and to read data as a byte stream. The client can always change these modes with DosSetPHandState.

An open can fail if all instances of the pipe are already open, or if the pipe has been closed by the client but is not yet disconnected by the server, or if no DosConnectNmPipe has been issued by the server since it disconnected the pipe.

After a pipe instance has been opened by a client, that same instance cannot be opened by another client at the same time. The opening process can, however, duplicate the handle with DosDupHandle to allow shared access to a pipe instance by a child process.

The access and sharing modes specified with DosOpen must be consistent with those specified by the server with DosMakeNmPipe.

Waiting for an Available Pipe Instance

If ERROR_PIPE_BUSY is returned by a DosOpen, a client process can issue DosWaitNmPipe and wait for a server on a pipe for which all instances are busy. The client waits a specified or default milliseconds value for an instance to become available. When a pipe instance becomes available, it is given to the waiting process whose thread has the highest priority.

Ending Access to the Named Pipe

Issuing **DosClose** ends the client's access to the named pipe. When all handles accessing one end of the pipe are closed, the pipe is considered broken. Thus, when the client end closes, no other client process can reopen the pipe until the server process issues a DosDisConnectNmPipe, followed by a DosConnectNmPipe.

Writing to a Named Pipe

Each write to a message pipe writes a message whose size is the length of the write. **DosWrite** and **DosWriteAsync** automatically encode the message length in the pipe.

If blocking mode is set for the message pipe, the write does not return until all requested bytes are written.

If nonblocking mode is set for a message pipe, and the size of the message is larger than the size of the pipe buffer, the write blocks. If the message size is smaller than the size of the buffer, but there is not enough room in the pipe at the time of the write, the write returns with no bytes written. No bytes are written in cases where the pipe is busy with another client.

If nonblocking mode is set for a byte pipe, and the size of the write is larger than the space available in the buffer at the time of the write, as many bytes are written as will fit in the buffer. The write returns with the number of bytes actually written.

An attempt to write to a pipe whose other end is closed returns ERROR_BROKEN_PIPE.

Reading a Named Pipe

Issuing **DosRead** or **DosReadAsync** with the handle of a named pipe returns data from the named pipe. A named pipe is read as one of the following:

- A byte pipe in byte stream mode
- A message pipe in message stream mode
- A message pipe in byte stream mode.

A byte pipe is read only in byte stream mode; however a message pipe can be read in either message stream or byte stream mode.

When a byte pipe is read, all currently available data up to the size requested is returned. If a message pipe is set to be read as a byte stream, the message headers are skipped over, and the pipe is read as if it were a byte pipe.

When a message pipe is set to be read as a message stream, a message is read in its entirety or not at all. If the length of a read is larger than the next available message, the message is read, and the BytesRead parameter indicates the size of the message read. If the read is smaller than the next available message, the read returns the number of bytes requested and ERROR_MORE_DATA. Subsequent reads block until the rest of the message can be transferred. DosPeekNmPipe may be used to determine how many bytes are left in the message.

When blocking mode is set, the read blocks until data is available. In this case the read never returns with BytesRead equals 0, unless end of file has been reached.

When nonblocking mode is set, the read attempts to read at the start of a message and returns if no message is available.

Synchronizing Pipe Dialogs

When used with named pipes, the file system function call **DosBufReset** performs an operation that is analogous to its defined one of forcing the buffer cache to disk. It blocks the calling process at one end of the pipe until all data it has written has been successfully read at the other end of the pipe. This allows communicating processes to synchronize their dialogs.

Transaction Processing on Duplex Message Pipes

The following function calls support transaction processing:

DosTransactNmPipe
: Performing a transaction on a duplex message pipe

DosCallNmPipe
: Performing a procedure on a duplex message pipe.

These calls are intended for use only on duplex message pipes. If one of these calls is issued for a pipe that is not a duplex message pipe, ERROR_BAD_FORMAT is returned.

Performing a Transaction

DosTransactNmPipe performs a transaction, a write followed by a read, on a message pipe. For this call to be successful, the pipe cannot contain any unread data and must be in message read mode.

The current setting of the blocking state has no effect on this call. DosTransactNmPipe writes the entire in-buffer to the pipe and does not return until it reads a response from the pipe into the out-buffer.

If the out-buffer is too small to contain the response message, ERROR_MORE_DATA is returned by the read.

Performing a Procedure

DosCallNmPipe has the combined effect of a DosOpen, DosTransactNmPipe, and a DosClose, and is referred to as a procedure call.

If an invalid pipe name is given with DosCallNmPipe, ERROR_PATH_NOT_FOUND is returned.

Getting Named Pipe Information

The following function calls support obtaining information about a named pipe:

DosPeekNmPipe
: Peeking at a pipe's contents and determining its state

DosQNmPHandState
: Determining the handle state set with DosMakeNmPipe or DosOpen, or reset with DosSetNmPHandState

DosQNmPipeInfo
: Querying actual buffer sizes and pipe instances amounts.

Peeking at the Pipe

DosPeekNmPipe is similar to a DosRead, except the bytes read from the pipe are not removed. In addition, the peek returns only that portion of the message currently in the pipe, even though the size of the peek can accommodate the entire message.

The peek never blocks, regardless of the blocking mode set. If the pipe cannot be accessed immediately, ERROR_PIPE_BUSY is returned.

DosPeekNmPipe also returns a value for the pipe's state. These values have the following meanings:

1 Disconnected
2 Listening
3 Connected
4 Closing.

If the pipe is in a disconnected state, this indicates that the server process has no client and is not ready to accept an open. A pipe is in a disconnected state immediately after:

- A DosMakeNmPipe, but before the first DosConnectNmPipe
- A DosDisConnectNmPipe, but before the next DosConnectNmPipe.

The server must issue DosConnectNmPipe to put the pipe in a listening state before a DosOpen can be accepted.

If the pipe is in a listening state, this indicates that the pipe can accept a DosOpen from a client. A pipe not in a listening state cannot be opened, and DosOpen returns ERROR_PIPE_BUSY.

If the pipe is in a connected state, this indicates that a client has opened the pipe and that both ends of the pipe have valid handles to perform I/O.

If the pipe is in a closing state, this indicates that the client end of the pipe has issued DosClose for all duplicates of the handle it has to the pipe, thus closing this end of the pipe. After it has successfully read all buffered data, the server end gets ERROR_EOF on reads and ERROR_BROKEN_PIPE on writes.

The server process must issue DosDisConnectNmPipe or DosClose to acknowledge the close of the client end. If a disconnect is issued, the pipe can be opened by the next client after a DosConnectNmPipe is issued by the server. If a close is issued, the pipe is deallocated.

Determining the Pipe Handle State

DosQNmPHandState returns information on the state of the pipe handle. If the handle is for the server end of the pipe, DosQNmPHandState returns information set with DosMakeNmPipe or reset with **DosSetNmPHandState**. If the handle is for the client end of the pipe, DosQNmPHandState returns information set with DosOpen or reset with DosSetNmPHandState.

DosSetNmPHandState can reset only the read mode (byte stream or message stream) and the pipe mode (blocking or nonblocking).

The read mode can be changed only if the pipe is a message pipe. Message pipes can be read either as byte streams or message streams; byte pipes can be read only as byte streams.

The pipe mode cannot be changed to nonblocking if another thread of the calling process is blocked on an I/O operation to the same end of the pipe.

Querying Actual Buffer Sizes and Pipe Instances

DosQNmPipeInfo returns level 1 file information for the following:

- Actual sizes of in-buffer and out-buffer
- Maximum number of pipe instances allowed
- Current number of pipe instances
- Length of pipe name
- ASCIIZ name of pipe, including *ComputerName*, if remote.

Coordinating Access to a Local Named Pipe

The following function calls support coordinating access to local named pipes:

DosSetNmPipeSem
: Attaching a system semaphore to a named pipe

DosQNmPipeSemState
: Getting the I/O state of pipes associated with a semaphore.

Attaching System Semaphores to Local Pipes

DosSetNmPipeSem attaches a system semaphore to a local named pipe. The semaphore is cleared when the pipe handle has either data to be read or write space available.

A semaphore can be attached to either side of the pipe, and up to two semaphores can be associated with a pipe, one for each side. If a semaphore is already attached to the side issuing DosSetNmPipeSem, the semaphore is overridden.

DosSetNmPipeSem provides support for server processes that need to handle a large number of incoming pipes. The process can issue a **DosSemWait** or a **DosMuxSemWait** on pipe semaphores to determine when I/O can be performed. By coordinating access to the server with semaphore calls, you avoid such resource-consuming methods as dedicating a thread for each incoming pipe or polling the pipes with nonblocking I/O.

A key handle enables a process, which has associated multiple pipes with a single semaphore by means of DosSetNmPipeSem, to distinguish events specific to a particular pipe when it issues DosQNmPipeSemState. The key handle is assigned an arbitrary value.

Note: DosSetNmPipeSem works only for local pipes. If an attempt is made to attach a semaphore to a remote pipe, ERROR_INVALID_FUNCTION is returned.

Getting the State of Local Named Pipes

A process normally uses **DosQNmPipeSemState** when it has associated a single semaphore with multiple pipes by means of DosSetNmPipeSem.

Issuing DosQNmPipeSemState places a record in the buffer for each local named pipe in the system if the specified system semaphore handle matches a semaphore attached to the named pipe, and:

- A blocking mode I/O operation can be started, or
- The pipe has been closed.

The format of a pipe record is:

BYTE Pipe status indicator that has the following values:
- **00H** No more records follow and rest of fields are undefined
- **01H** Read data is available
- **02H** Write space is available
- **03H** Pipe is closed.

BYTE Contains pipe flags that indicate:
- **01H** A thread is waiting at the other end of pipe.

WORD Contains key value specified by SemHandle when DosSetNmPipeSem was issued.

WORD Write space or read data amount, depending on value of *pipe status.*

There is no guarantee that records in the buffer refer to named pipes opened by the process that has issued DosQNmPipeSemState. If the same system semaphore is attached to different named pipes by multiple processes, information about named pipes that are inaccessible can be returned. Cooperating processes can agree on a convention for creating key values to help distinguish the attached process.

A process that wants data in the buffer to refer only to its own named pipes can use an exclusive system semaphore by issuing DosCreateSem.

Part 8. Printing and Device Support

Chapter 36. Printing

Your application can send alphanumeric and graphic data directly to a printer for immediate printing, or it can send the data to the *spooler* for printing when a suitable printer is free. The spooler adds each print job to one of the queues that are associated with your printers.

This chapter describes the spooler and its components, and shows the benefits of using the spooler in a multitasking environment. Detailed descriptions are given for:

- How to print alphanumeric data
- How to print graphic data using the device function calls
- How to print graphics text
- How to print a bit map of the screen.

The differences when sending data direct to a printer are described in "Printing Without the Spooler" on page 36-13.

The term *printer* is used in this chapter to describe any hard-copy output device.

The Spooler

The spooler manages printers. Data sent to these printers is diverted to a *spool file* and each spool file represents a complete print job. The spooler submits the spool file for printing when the file is complete and when a suitable printer is available.

The spooler has several components; the term *spooler* is used when referring to these components as a group. The principal components of the spooler are described in this chapter.

Benefits of Using the Spooler

There are many advantages in using the spooler to control printing. The main ones are:

- The spooler prevents more than one application writing to a printer at one time.
- The spooler supports a number of printers simultaneously. The printers attached to a system can be shared among all the applications running on the system.
- The spooler is responsible for managing the queues associated with the printers, and for printing the spool files on an appropriate device. This is done in the background, without interfering with other applications that are running on the system.
- The spooler can improve performance. Some devices can be slow to print, so you should allow the spooler to select the best time and the best device for printing your job. This frees your applications to continue their processing. While a file is waiting to be printed, it is not occupying memory.
- The spooler can handle *form names*. A form name identifies the paper on which you want to print. You can submit a print job with a request for a particular form without having to ensure that the form is loaded. The spooler holds the job until the required form is loaded.
- To the application, the interface to the spooler is very similar to the interface to the display. The mechanics of the printing process can be entirely hidden from the application.
- With the right choice of spool-file format, application programs can be device-independent. This is because the presentation driver can process the output data into a format suitable for the printer selected by the spooler.

The spooler gives the user flexibility in organizing and optimizing the use of the system's printers. It has a global view of the system's printing resources, and is therefore able to make the best use of them. An application program does not have the same privileged view of the system and thus cannot use the system's resources as effectively.

A single queue can be associated with more than one printer, as shown in Figure 36-1 on page 36-2.

When printers are pooled in this way, the spooler is able to ensure that the printers are evenly occupied. The printers could have different types of paper loaded, but you need only one queue because the spooler is able to select the correct printer for each job.

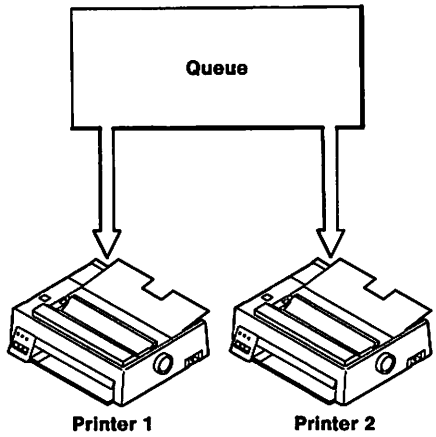


Figure 36-1. One Queue Associated With More Than One Printer

More than one queue can be associated with a single printer, as shown in Figure 36-2.

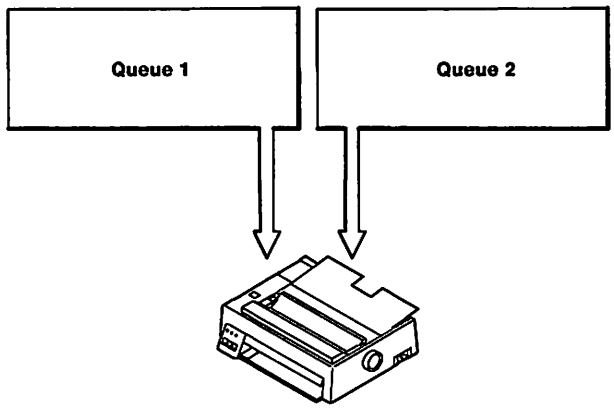


Figure 36-2. More Than One Queue Associated With a Single Printer

The advantage of this printer configuration is that you can assign priorities to your queues to get the best use from the printer. For example, queue 1 could contain short print jobs that are needed quickly. This queue could have exclusive use of the printer at the busiest times of the day. Queue 2 could contain longer print jobs, and be placed on hold until queue 1 is empty or less busy. Another example is the use of separate queues for draft printing and for near-letter-quality printing. This removes the need to change the printer's mode repeatedly. You could print, from the first queue, all the jobs that require draft mode, then change the printer to near-letter-quality mode and print the jobs in the second queue.

A good configuration is the association of many queues with many printers, as shown in Figure 36-3. In this arrangement, queue 1 could be used for near-letter-quality printing and queue 2 for draft-quality printing. This allows the spooler to make use of printer 1 for draft-quality printing when printer 2 is busy. Queue 3 could be associated with a printer that is situated in a locked room, and so could be used for confidential printing. Such a configuration makes good use of the spooler, and gives the best turn-around times for print jobs.

Overview of the Spooling Process

The route that data can follow from your application, through the spooler, to the printer is shown in Figure 36-4 on page 36-4. This is not the only possible route, but it is the one you will use most often. The route to the printer for *direct* printing is also shown.

The steps in the spooling process are:

1. Your application creates a *queued* device context and sends data to it. Chapter 37, "Creating a Printer Device Context" describes how to do this. The presentation driver (which you specify when you create the device context) writes the data to a spool file.

2. The Print Manager creates a print job by associating a control file with each spool file. The control file contains information such as the format of the spool file and the number of copies required. The Print Manager writes the spool file to a disk, and assigns it to the selected queue.

 The Print Manager window is the user interface for the spooler. It allows the user to administer the queues dynamically. For example, jobs that are queued for printing can be held, canceled, or repeated.

3. When a job is ready for printing, the spooler queue processor is invoked by the Print Manager. The queue processor reads the spool file from the queue and passes the data to the presentation driver. Your application can let the queue processor control the way that the data is printed if the data is not in a device-specific format.

4. The presentation driver passes the spool file to the kernel device driver, from where it is sent to the printer.

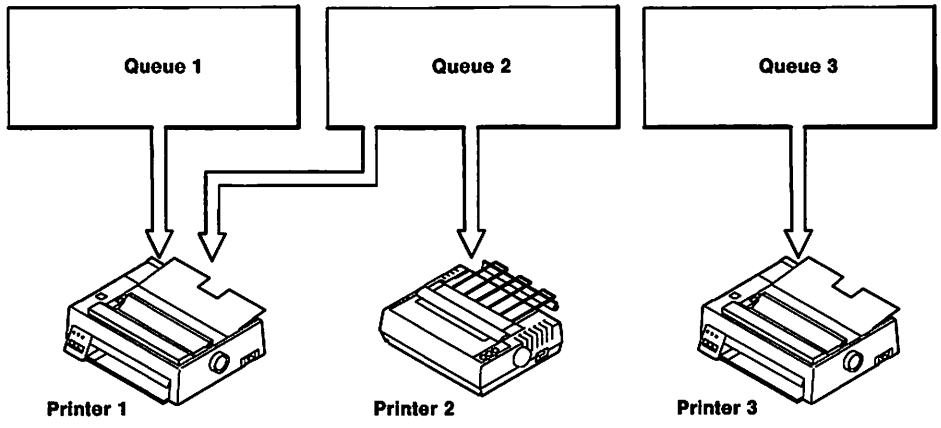


Figure 36-3. Many Queues Associated With Many Printers

Spool-File Formats

Presentation Manager supports two standard spool-file formats, PM_Q_RAW and PM_Q_STD.

PM_Q_RAW Format

Raw data is data that is generated by an application for use by a particular printer. A simple example of raw data is an ASCII file containing alphanumeric and printer-control characters only. Spool files containing this type of data are in PM_Q_RAW format.

Because the printer data stream is generated by your application, you cannot change the appearance of the output using any of the spooler controls, such as the queue-processor parameters. This means that PM_Q_RAW format is less suitable for spooled graphic data.

Because PM_Q_RAW data is device-dependent, it gives different outputs on a range of printers. The spooler is not used to its greatest advantage because output is restricted to a particular set of printers. These printers may not always be available when they are needed.

PM_Q_STD Format

PM_Q_STD-format spool files are used to hold graphic data. This is any data generated using graphics calls, including graphics text. A presentation driver can translate these calls and write the data to a spool file. You can include raw data in a PM_Q_STD-format spool file by using driver-specific escape codes.

PM_Q_STD is the most versatile of the spool-file formats, and is the one you are recommended to use. PM_Q_STD-format data is likely to give similar output on a range of printers. This frees the spooler to make the optimum use of your system.

The advantage of PM_Q_STD format over PM_Q_RAW format can be shown by reference to Figure 36-3. If queue 2 has a PM_Q_RAW-format spool file that contains data formatted for printer 1, the job has to wait until printer 1 is free, even if printer 2 is idle. However, if the spool file is in PM_Q_STD format, the spooler can direct the file to printer 2 if printer 1 is busy, because the presentation driver formats the data for the printer that the spooler selects.

The Role of the Presentation Driver

The presentation driver has two functions:

1. It answers queries from applications about the capabilities of the physical device with which it is associated.
2. It formats and transmits data. Some presentation drivers reset the printer to a state similar to its power-on state before each job is printed.

The first function is to provide the answers when an application issues any of the following query-type calls:

DevQueryDeviceNames (WDQDN)

This call returns a list of the names of the devices supported by the driver, together with descriptive information about the devices, and information about the spool-file formats supported by the driver.

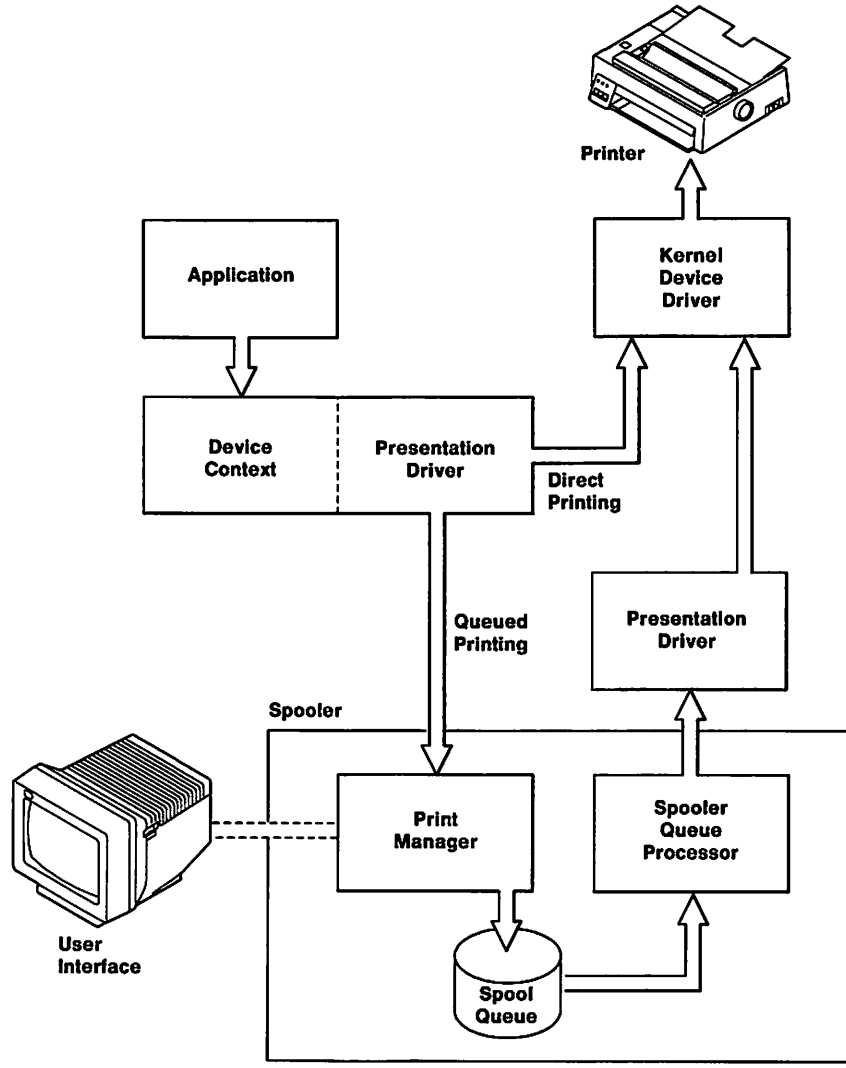


Figure 36-4. The Printing Process

DevQueryCaps (WDQCAP)
This call returns information about the capabilities and the characteristics of the device. For example, you can use this call to find out whether it is a raster device or a vector device, and whether it supports banding. You can also use it to get information about its graphics support, such as the number of distinct colors supported in the physical palette of the device, or the number of device-specific fonts supported.

DevQueryHardcopyCaps (WDQHC)
This call returns information such as the clip limits of the device and the name of the current form.

DevEscape (WDESC)
This call allows your application to get further information about a device. The following query functions are defined:

DEVESC_QUERYESCSUPPORT
Tells you whether a particular escape code is supported by the presentation driver.

DEVESC_GETSCALINGFACTOR
Retrieves the scaling factors for the x and y axes of a printing device.

DEVESC_QUERYVIOCELLSIZES
Queries the VIO cell sizes supported by the presentation driver.

For more information about these calls, see the *Programming Reference: Volume 1.*

The presentation driver performs its second function (formatting and transmitting data) at two different stages in the spooling process.

1. The driver is first involved when an application defines a queued device context. It formats PM_Q_STD data and writes it to a spool file. Data in other formats is written straight to the spool file. The driver then passes the spool file to the Print Manager.
2. The driver is involved again later in the spooling process when it is invoked by the spooler queue processor. PM_Q_STD data, in the form of graphics orders, are translated and passed to the kernel device driver ready for use by the physical device. PM_Q_RAW data, received as 'RawData' escapes, are passed directly to the kernel device driver.

For more information about presentation drivers, see *I/O Subsystems and Device Support: Volume 2.*

The Role of the Print Manager

The Print Manager is the main component of the spooler. It performs these tasks:

1. It receives formatted data from the presentation driver and writes it to a spool file on disk.
2. It makes a print job of each spool file and assigns it to the selected queue.
3. It invokes a spooler queue processor to handle each print job when it is ready for printing.

Using the Print Manager window, the user can:

- Hold, and subsequently release, a job. A job that is held cannot be printed.
- Hold a queue to prevent any of its jobs being printed, and release the queue again when ready.
- Move a job to the head of a queue. The user can also change the priority number of any job.
- Cancel a single job or all the jobs in a queue.
- Repeat a job so that it appears twice in the same queue.
- Stop and restart a job.

You can find more information about the user interface to the Print Manager in *Getting Started.*

How the Print Manager Selects a Job for Printing

The Print Manager has its own algorithm for managing the queues. When it adds a print job to a queue, it positions the job before all jobs with a lower priority, but after all jobs with an equal or higher priority. You can specify the priority of your print job when you create it (see **spoolerparams** under "The Elements of the DEVOPENDATA Structure" on page 37-2).

The Print Manager uses the following rules to determine when a job should be made available to a queue processor for printing:

1. There may be a number of queues for a single printer. Their priorities are in the order in which the user defined them to the spooler. The Print Manager examines the first queue for the printer to find those jobs that have both the highest priority and the current (or a selectable) form name. The form name, like the print-job priority, is specified on the DevOpenDC (WDOPEN) call (see **spoolerparams** under "The Elements of the DEVOPENDATA Structure" on page 37-2). A job that has no form name specified is considered to have the current form name, and is printed using the current form. Any job or queue currently on hold is not eligible for printing.
2. If more than one job has both the highest priority and the correct form name, the job that is nearest to the head of the queue is printed first.
3. If no jobs in the first queue satisfy the priority and form-name requirement, the second queue for the printer is examined, and so on, through all of the queues.

The Role of the Spooler Queue Processor

When a job satisfies the job-selection algorithm, the Print Manager invokes a spooler queue processor to print it. There is a default queue processor for each queue. This is identified when the user defines the queue to the spooler.

The queue processor takes the print job from the queue. If the data is in PM_Q_STD format, the queue processor can change the appearance of the output, using the optional queue-processor parameters. These are specified by the application when the print job is created (see **queueprocparams** under

"The Elements of the DEVOPENDATA Structure" on page 37-2). The parameters are described in "Passing Parameters to the PMPRINT Queue Processor" on page 37-7. The queue processor then submits the data to the presentation driver of the chosen printer.

The Presentation Manager-supplied spooler queue processor, PMPRINT, supports the standard spool-file formats. PMPRINT is intended for use with raster printers, although it can be used with vector plotters.

Printing Alphanumeric and Other Raw Data

Raw data can be printed in two ways:

1. Using the device function calls
2. Using the Dos function calls.

The **device API** method is recommended because it allows your application to control the print job when it is spooled. For example, you can:

- Choose a queue
- Specify the priority of the print job
- Query the capabilities of a printer
- Select features such as print quality and named forms
- Control the way that the queue processor handles the print job
- Format the output (such as force a new page) without knowing how to program a particular printer
- Mix graphic and raw text data, although not on the same page
- Give a description of the print job for the Print Manager to display to the user.

Also, you can specify *direct* printing, which bypasses the spooler. You may want to do this for reasons of security.

For the printing of ASCII text, the **Dos API** calls provide a simple method of creating output. However, output must be directed to a specific device address (such as LPT1) and your application has no control over the print job.

Using the Device Function Calls

To use the device function calls to print data, you must first create a *device context* for the printer. This is a logical description of the physical device and it allows you to query the capabilities of the device. Follow these steps:

1. Select an output device (see "Selecting the Output Device" on page 37-3) and, if desired, select a form (see "Getting Form Selections and Job Properties" on page 37-4).
2. Use the DevOpenDC (WDOPEN) call to create a device context (see "The DevOpenDC (WDOPEN) Call" on page 37-1).
3. Use the DevEscape (WDESC) call with the DEVESC_STARTDOC code to signal the start of the document. When you use the DEVESC_STARTDOC code, you can give your document a name. The name is displayed in the Print Manager window, together with the other job details, when the job is queued.
4. Use the DevEscape (WDESC) call with the DEVESC_RAWDATA code to write the data to the device context.
5. Use the DevEscape (WDESC) call with the DEVESC_NEWFRAME code to start a new page. There is no need to use the DEVESC_NEWFRAME code for the first page.
6. Repeat steps 4 and 5 as required. Do not use the DEVESC_NEWFRAME code at the end of the last page.
7. When you have finished creating the document, use the DevEscape (WDESC) call with the DEVESC_ENDDOC code to signal this. You can repeat the procedure from step 3 to create any number of print jobs.

 All the data between the DEVESC_STARTDOC and DEVESC_ENDDOC codes is processed as one file by the spooler and becomes one print job.
8. When you have no more data to send, use the DevCloseDC (WDCLS) call to close the device context.

An example of this method is shown in "Example Code: Printing Using Device Calls" on page 36-16.

Using the Dos Function Calls

If your application produces only alphanumeric data, you can print it using the Dos function calls. The steps are:

1. Use the DosOpen call to start the job, giving the address of the printer you want to use (for example, LPT1) for the **filename** parameter. If the user has the spooler selected, DosOpen opens a spool file; otherwise, it opens the printer for direct printing.

 The following code shows how to use this call:

```
HFILE  hfPrt;
USHORT rc;
USHORT usAction;
rc = DosOpen( "LPT1" /* File name          */
   , &hfPrt    /* File handle (returned)  */
   , &usAction /* Action taken (returned) */
   , 0         /* Initial file size       */
   , 0         /* File-attribute bits     */
   , 0x0001    /* File-open flag          */
   , 0x60C1    /* File-open mode          */
   , 0L        /* Reserved, must be zero  */
   );
```

 The file-open flag instructs OS/2 to open the spool file or to open the printer. The file-open mode sets flags that control options such as read/write access.

2. Use the DosWrite call to write your data. If the user has the spooler selected, the data is written to the spool file; otherwise, it is written directly to the printer. Specify the file handle that is returned by DosOpen.

3. Use the DosClose call to end the job. Specify the file handle that is returned by DosOpen. If the user has the spooler selected, DosClose closes the spool file and writes the file to disk. The spooler creates one print job for each pair of DosOpen and DosClose calls. If the spooler is not selected, DosClose closes the printer after all the data has been sent. Your application must wait until then.

Data produced in this way is passed directly to the kernel device driver from your application. The kernel device driver has a *spooler monitor* that intercepts the data and passes it to the Print Manager.

Printing Graphic Data

Graphic data is the most general form of output data. It includes lines, filled areas, text, and images. Even if your data contains text only, you can use graphics to enhance its presentation. Features such as proportional fonts and various font sizes are available.

Note: You can use the Presentation Manager utility *Print Picture* if you want to print an existing metafile. See "Printing Graphics Using Print Picture" on page 36-9.

The basic steps when printing graphic data are very similar to those described above for printing alphanumeric data using the device function calls. In addition, you must associate the graphics presentation space in which you prepare your data with the printer device context. Follow these steps:

1. Use the GpiCreatePS (GIPCRT) call to create a graphics presentation space (see Chapter 19, "The Graphics Presentation Space"). You will create the device-independent description of the graphic data in this presentation space.

2. Select an output device (see "Selecting the Output Device" on page 37-3) and, if desired, select a form (see "Getting Form Selections and Job Properties" on page 37-4).

3. Use the DevOpenDC (WDOPEN) call to create a device context (see "The DevOpenDC (WDOPEN) Call" on page 37-1).

4. Use the GpiAssociate (GSASS) call to associate the graphics presentation space with the device context.

5. Use the DevEscape (WDESC) call with the DEVESC_STARTDOC code to signal the start of the document. When you use the DEVESC_STARTDOC code, you can give your document a name. The name is displayed in the Print Manager window, together with the other job details, when the job is queued. This is shown in "Example Code: Printing Using Device Calls" on page 36-16.

6. Create a page of output using any combination of the following calls:

 - Any graphics-primitive drawing request, such as GpiLine (GILINE) or GpiCharString (GSCHAP)
 - GpiDrawSegment (GSDSEG)
 - GpiDrawFrom (GSDF)

- GpiDrawChain (GSDCHN)
- GpiPlayMetafile (GSLOAD).

If you want to include alphanumeric data, you can supply it as input to the GpiCharString (GSCHAP) call. You can also produce a page of output using the DevEscape (WDESC) call with the DEVESC_RAWDATA code, but you must not mix raw data and graphics calls on a single page.

7. Use the DevEscape (WDESC) call with the DEVESC_NEWFRAME code to start a new page. There is no need to use the DEVESC_NEWFRAME code for the first page.
8. Repeat steps 6 and 7 as required. Do not use the DEVESC_NEWFRAME code at the end of the last page.
9. When you have finished creating the document, use the DevEscape (WDESC) call with the DEVESC_ENDDOC code to signal this. You can repeat the procedure from step 5 to create any number of print jobs.

 All the data between the DEVESC_STARTDOC and DEVESC_ENDDOC codes is processed as one file by the spooler and becomes one print job.
10. When you have no more data to send, use the GpiAssociate (GSASS) call with a null device-context handle to disassociate the presentation space from the device context.
11. Use the GpiDestroyPS (GIPDEL) call to destroy the presentation space, and the DevCloseDC (WDCLS) call to close the device context.

These steps are illustrated in the following example. It shows a sequence of calls that produces a four-page PM_Q_STD spool file:

```
GpiCreatePS
     /* Create a presentation space          */
PrfQueryProfileString
     /* Get the printer name.                */
PrfQueryProfileString
     /* Get the presentation driver and      */
     /* queue name for the printer.          */
DevOpenDC
     /* Create a queued or direct            */
     /* device context.                      */
GpiAssociate
     /* Associate the presentation space     */
     /* with the device context.             */
DevEscape(DEVESC_STARTDOC)
     /* Start the print job.                 */
GpiDrawChain
     /* Draw page 1.                         */
DevEscape (DEVESC_NEWFRAME)
     /* Request form feed.                   */
GpiDrawFrom
     /* Draw page 2.                         */
DevEscape (DEVESC_NEWFRAME)
     /* Request form feed.                   */
GpiPlayMetafile
     /* Draw page 3.                         */
DevEscape (DEVESC_NEWFRAME)
     /* Request form feed.                   */
GpiSetCurrentPosition
     /* Set the start position for drawing   */
     /* nonretained graphics on page 4.      */
GpiPolyline
GpiBox
GpiCharStringAt
DevEscape (DEVESC_ENDDOC)
     /* End the print job                    */
GpiAssociate
     /* Use a null pointer to disassociate   */
     /* the presentation space from the      */
     /* device context.                      */
GpiDestroyPS
     /* Destroy the presentation space       */
DevCloseDC
     /* Close the device context.            */
```

Printing Graphics Text in Text Mode

Presentation Manager allows you to use a range of fonts and font sizes to display and print character data. It allows you to mix the text with graphics, and to customize the output using techniques such as character shearing and stretching (these are described in Chapter 23, "Graphics Text"). For Presentation Manager to offer these features, it uses your output devices in graphics mode. Many printers have low output rates when in graphics mode because a lot of processing is involved for each band of graphics data. However, if your data consists of characters only, and you do not want to customize the output in any way, you can tell Presentation Manager to use your printer in text mode to give the fastest possible output.

The degree to which character-output performance is improved when a device is used in text mode depends on the presentation driver. The driver can choose to:

- Ignore graphics primitives such as lines, arcs, and areas
- Approximate the positioning of text to the nearest character position that the device supports for the current font
- Use the fonts provided by the device.

To print graphics text using text mode, follow these steps:

1. Create a graphics presentation space and a printer device context, as described in "Printing Graphic Data" on page 36-7.
2. Use the GpiAssociate (GSASS) call to associate the graphics presentation space with the device context.
3. Use the DevEscape (WDESC) call with the DEVESC_QUERYESCSUPPORT code to test whether the presentation driver supports text-mode printing. Test for the DEVESC_DRAFTMODE function.
4. Use the DevEscape (WDESC) call with the DEVESC_STARTDOC code to signal the start of the document.
5. Use the DevEscape (WDESC) call with the DEVESC_DRAFTMODE code to turn on draft mode.
6. Use the GpiQueryFonts (GSQF) call to query what fonts are available.

 Note: If you are printing using the spooler, and have requested a PM_Q_STD-format spool file, you can use only public fonts. Use the QF_PUBLIC option on the GpiQueryFonts (GSQF) call to query only the public fonts. For more information, see "Public and Private Fonts" on page 23-2.
7. Use the GpiCreateLogFont (GSCRLF) call to select a font. Set the FATTR_FONTUSE_NOMIX flag in the FATTRS structure to show that the output data contains text only. Do not set the OUTLINE or the TRANSFORMABLE font-use flags.
8. Use the GpiSetCharSet (GSCS) call to identify the logical font (defined in step 7) as the current font.
9. Create a page of output using any combination of the graphics-text functions. The number of functions that are available is driver-dependent. However, all drivers should allow you to use the following:
 - GpiCharString (GSCHAP)
 - GpiCharStringAt (GICHAR)
 - GpiCharStringPos (GICHPP)
 - GpiCharStringPosAt (GICHPR)
 - GpiSetCurrentPosition (GICP)
 - GpiMove (GIMOVE).
10. Use the DevEscape (WDESC) call with the DEVESC_NEWFRAME code to start a new page. There is no need to use the DEVESC_NEWFRAME code for the first page.
11. Repeat steps 9 and 10 as required. Do not use the DEVESC_NEWFRAME code at the end of the last page.
12. When you have finished creating the document, use the DevEscape (WDESC) call with the DEVESC_ENDDOC code to signal this. You can repeat the procedure from step 4 to create any number of print jobs.
13. Use the DevEscape (WDESC) call with the DEVESC_DRAFTMODE code to turn off draft mode.
14. When you have no more data to send, disassociate and destroy the presentation space, then close the device context, as described in "Printing Graphic Data" on page 36-7.

Printing Graphics Using Print Picture

You can print a picture that is defined in a metafile or in a picture interchange format (PIF) file using the Presentation Manager utility called **Print Picture**. If you supply a PIF file, Print Picture uses the Presentation Manager utility **Convert Picture File** to convert it to a metafile before printing it.

Note: Print Picture does not work when the spooler is not selected.

Print Picture has a user interface which you can invoke from the Desktop Manager window. The interface allows you to:

- Select the file(s) that you want to print
- Select a printer

- Select the form that you want to print on
- Select the number of copies
- Select color or monochrome output
- Set the size and position of the output area on the form.

The last four items in this list define the queue-processor parameters that are described in "Passing Parameters to the PMPRINT Queue Processor" on page 37-7. When you set the options in the Print Picture window, Print Picture sets the corresponding queue-processor parameters.

The user interface allows you to view the picture before you print it.

You can also invoke Print Picture from the OS/2 command prompt. Type the command:

```
PicPrint [[d:][path]filename.ext [/P]]
```

The file name is the name of the picture file that you want printed. If the file is in picture interchange format, use the .PIF file-name extension. If the file-name extension is anything other than .PIF, Print Picture acts as if the file is a metafile. The /P option causes the file to be printed using settings from the Print Picture window. If you omit this option, the Print Picture window is displayed, with the specified file selected. If you supply no parameters at all for the Print Picture command, the Print Picture window is displayed, but no file is selected.

For more information on the Print Picture user interface, see *Getting Started.*

Invoking Print Picture from Your Application

You can print metafiles and PIF files from within your application using the PicPrint (PIPRT) call. You supply the name and type (metafile or PIF) of the file as parameters to the call. There are a further six optional parameters:

COP Gives the number of copies of the spool file that you want printed

ARE Specifies the size and position of the output area on the printed page

FIT Determines which part of the picture is to be printed

COL Specifies color or monochrome output

MAP Determines how the neutral colors (those that are not specified in the picture file) are printed.

FORM Gives the name of the form on which you want the picture printed.

The first five parameters are the same as those for the PMPRINT queue processor. When you specify their values in the PicPrint (PIPRT) call, Print Picture sets the corresponding queue-processor parameters. The queue-processor parameters are described in detail in "Passing Parameters to the PMPRINT Queue Processor" on page 37-7.

The full syntax for the PicPrint (PIPRT) call is described in the *Control Program Programming Reference.*

Printing a Copy of the Screen

A convenient way of printing a copy of the screen, or part of it, is to copy the data to memory as a *bit map*, then copy the bit map to the printer.

Follow these steps to create a bit map of the screen:

1. Use the DevOpenDC (WDOPEN) call to create a memory device context that is compatible with the screen (see "The DevOpenDC (WDOPEN) Call" on page 37-1). Specify a driver name of DISPLAY in the DEVOPENDATA structure.
2. Use the GpiCreatePS (GIPCRT) call to create a presentation space and associate it with the device context. Pass the handle of the device context as a parameter to this call and specify the GPIA_ASSOC option. You must specify pels for the units of the presentation-page size.
3. Use the GpiCreateBitmap (GSCBM) call to create a bit map for the memory device context.
4. Use the GpiSetBitmap (GSCBM) call to select the bit map.
5. Use the WinGetScreenPS (WPGSPS) call to get the handle of the screen presentation space.
6. Use the GpiBitBlt (GIBB) call to copy the bit map from the screen presentation space to the screen-compatible memory presentation space.
7. Use the GpiSetBitmap (GSSBM) call with a null handle to deselect the bit map.
8. Use the GpiAssociate (GSASS) call with a null device-context handle to disassociate the presentation space from the device context.
9. Use the GpiDestroyPS (GIPDEL) call to destroy the presentation space.
10. Use the DevCloseDC (WDCLS) call to close the device context.

The sequence of calls is:

```
DevOpenDC (OD_MEMORY, "DISPLAY")
     /* Create a memory device context,  */
     /* compatible with the screen.      */
GpiCreatePS (GPIA_ASSOC | PU_PELS)
     /* Create a presentation space and  */
     /* associate it with the device     */
     /* context.                         */
GpiCreateBitmap
     /* Create a bit map.                */
GpiSetBitmap
     /* Select the bit map.              */
WinGetScreenPS
     /* Get the handle of the screen     */
     /* presentation space.              */
GpiBitBlt
     /* Copy the bit map from the screen */
     /* presentation space to the memory */
     /* presentation space.              */
GpiSetBitmap (NULL)
     /* Deselect the bit map.            */
GpiAssociate (NULL)
     /* Disassociate the presentation    */
     /* space from the device context.   */
GpiDestroyPS
     /* Destroy the presentation space.  */
DevCloseDC
     /* Close the device context.        */
```

This procedure is described in greater detail in "Creating and Drawing Bit Maps" on page 24-2.

The following steps show one way to print the bit map:

1. Use the DevOpenDC (WDOPEN) call to create a printer device context (see "The DevOpenDC (WDOPEN) Call" on page 37-1).
2. Use the GpiCreatePS (GIPCRT) call to create a presentation space and associate it with the printer device context. Pass the handle of the device context as a parameter to this call and specify the GPIA_ASSOC option.
3. Use the DevOpenDC (WDOPEN) call to open a printer-compatible memory device context. Set the **comp** parameter to the handle of the printer device context that you created in step 1.
4. Use the GpiCreatePS (GIPCRT) call to create a presentation space and associate it with the memory device context. Pass the handle of the device context as a parameter to this call and specify the GPIA_ASSOC option.
5. Use the GpiSetBitmap (GSSBM) call to select the bit map.
6. Use the GpiBitBlt (GIBB) call to copy the bit map from the memory presentation space to the printer presentation space.
7. Use the GpiSetBitmap (GSSBM) call with a null handle to deselect the bit map.
8. Use the GpiAssociate (GSASS) call with a null device-context handle to disassociate the memory presentation space from its device context.
9. Repeat this for the printer presentation space.
10. Use the GpiDestroyPS (GIPDEL) call to destroy the memory presentation space.
11. Repeat this for the printer presentation space.
12. Use the DevCloseDC (WDCLS) call to close the memory device context.
13. Repeat this for the printer device context.
14. Use the GpiDeleteBitmap (GSDBM) call to delete the bit map.

The sequence of calls is:

```
DevOpenDC
     /* Create a queued or direct        */
     /* printer device context.          */
GpiCreatePS (GPIA_ASSOC)
     /* Create a presentation space and  */
     /* associate it with the printer    */
     /* device context.                  */
DevOpenDC (OD_MEMORY)
     /* Create a memory device context   */
     /* that is compatible with the      */
     /* printer.                         */
GpiCreatePS (GPIA_ASSOC)
     /* Create a presentation space and  */
     /* associate it with the memory     */
     /* device context.                  */
GpiSetBitmap
     /* Select the bit map.              */
GpiBitBlt
     /* Copy the bit map from the memory */
     /* presentation space to the        */
     /* printer presentation space.      */
GpiSetBitmap (NULL)
     /* Deselect the bit map.            */
GpiAssociate (NULL)
     /* Disassociate the memory          */
     /* presentation space from its      */
```

```
    /* device context.                    */
GpiAssociate (NULL)
    /* Disassociate the printer           */
    /* presentation space from its        */
    /* device context.                    */
GpiDestroyPS
    /* Destroy the memory presentation    */
    /* space.                             */
GpiDestroyPS
    /* Destroy the printer presentation   */
    /* space.                             */
DevCloseDC
    /* Close the memory device context.   */
DevCloseDC
    /* Close the printer device context.*/
GpiDeleteBitmap
    /* Delete the bit map.                */
```

Using a Separate Thread for Printing

To ensure that the system always remains responsive to the user, you should use a separate thread to handle any lengthy or processor-intensive operations. Printing is such an operation. The segment of code in "Example Code: Printing Using a Separate Thread" on page 36-14, taken from the AVIOSAMP.C sample program, shows how to create, and then kill, a separate thread. Further information on writing multithreaded applications is given in "Multithreading Considerations for PM Applications" on page 34-4.

Handling Print Errors

Errors can be generated by the printer subsystem *after* your application has created a print file. The Print Manager and the presentation driver are responsible for displaying message boxes to the user if you are printing through a queued device context. For direct printing however, your application must have a *message queue* on its printing thread to allow the presentation driver to handle errors such as the printer being off line or out of paper.

This is demonstrated in the code segment shown in "Example Code: Printing Using a Separate Thread" on page 36-14, based on the AVIOSAMP.C sample program. Additions to the source code are shown in **bold** print. The PRINTTHREAD() function calls the PRINTOUT() function, part of which is listed in "Example Code: Printing Using Device Calls" on page 36-16.

Stopping the Creation of a Print Job

An example of a situation where your application must respond to the user is the cancellation of a print request. When creating a print job using the device function calls, you can use the DevEscape (WDESC) call with the DEVESC_ABORTDOC code to stop the print job before the spool file is created. You cannot stop a print job after you have issued the DevEscape (WDESC) call with the DEVESC_ENDDOC code. The DEVESC_ABORTDOC code is used in the code segment that is shown in "Example Code: Printing Using Device Calls" on page 36-16.

Using the Spooler Directly

The spooler function calls are provided to allow your application to write data direct to a spool file. This means that the data by-passes the presentation driver, so it must be in a format that the printer can understand. Your application must format the data.

The advantage of this method is that there is no need to create a device context, so it is slightly faster than printing using the device function calls. A disadvantage is that if the user does not have the spooler selected, printing cannot occur.

An example of when you can invoke the queue manager directly is when you are writing alphanumeric data to a PM_Q_RAW-format spool file.

Follow these steps to create a spool file:

1. Use the SplQmOpen (SMOPEN) call to open the Print Manager. This call is similar to the DevOpenDC (WDOPEN) call (see "The DevOpenDC (WDOPEN) Call" on page 37-1). Its **data** parameter has the same structure as that for DevOpenDC. You must specify a *queue name* (for example, LPT1Q) for the address element of the DEVOPENDATA structure.
2. Use the SplQmStartDoc (SMSDOC) call to signal the start of your document. This is similar to the DevEscape (WDESC) call with the DEVESC_STARTDOC code that you use when printing with the device function calls.

3. Write data to the spool file using the SplQmWrite (SMWRT) call.
4. Use the SplQmEndDoc (SMEDOC) call to signal the end of your document. This is similar to the DevEscape (WDESC) call with the DEVESC_ENDDOC code that you use when printing with the device function calls.
5. Use the SplQmClose (SMCLS) call to close the Print Manager. This call is similar to the DevCloseDC (WDCLS) call.

Printing Without the Spooler

You may want to force printing from your application to be done without the spooler. For example, you may have a dedicated printing device that should not be shared with other applications.

The only differences to your application code in this case are in the way that you create the printer device context.

- Specify a *direct* device context in the **type** parameter of the DevOpenDC call (see "The DevOpenDC (WDOPEN) Call" on page 37-1).
- Specify the printer port name (for example LPT1) for the **address** element of the **data** parameter of the DevOpenDC call.
- If you give any spooler or queue-processor parameters in the **data** parameter of the DevOpenDC call, they are ignored.

What if the Spooler is Not Selected?

You may code your application with the intention of letting the spooler manage the printing. However, the user may not have the spooler selected when the application is run. In this case, data is sent direct to the printer via a *direct* device context. This means that any spooler or queue-processor parameters that you have specified are ignored. In particular, graphics output will be different if you had specified a "scaled to fit" print because the default format when the spooler is not selected is "real size."

The spooler function calls do not operate when the spooler is not selected. Therefore, if you use these in your application, it does not produce any printed output when the spooler is not selected.

Example Code: Printing Using a Separate Thread

These code segments are taken from the AVIOSAMP.C sample program. They show how to create and kill a print thread, and how to use a message queue on the print thread. Changes that have been introduced to the sample program to show the use of message boxes are shown in **bold** print.

```
#define  STACKSIZE  4096
#define  PRINTDOC   WM_USER+1            /* Print contents of VIO     */
                                         /* presentation space        */
#define  KILLTHREAD WM_USER+2            /* Terminate print thread    */
VOID FAR PrintThread (VOID);

HAB      habPrn;                   /* Print thread anchor block handle */
HMQ      hmqPrn;                         /* Print thread queue handle */
SEL      selStack;
ULONG    KillThreadSem  = 0;             /* Semaphore to signal that  */
HSEM     hsemKillThread = &KillThreadSem; /* thread has ended.        */
BOOL     fPrinting = FALSE;

/*************************************************************************/
/* CREATETHREAD:                                                         */
/* Allocate a stack for the secondary thread, and set it running.        */
/* All printing is done in the background from within the secondary      */
/* thread.                                                               */
/*************************************************************************/
VOID CreateThread()
{
  TID tidPrn;

  DosAllocSeg( STACKSIZE, &selStack, 0 );
  DosCreateThread( PrintThread, &tidPrn, MAKEP( selStack, STACKSIZE-2 ));
}

/*************************************************************************/
/* KILLTHREAD:                                                           */
/* Close down the secondary print thread.  Wait on the semaphore to      */
/* indicate that the print thread has ended, before freeing its stack.   */
/*************************************************************************/
VOID KillThread()
{
  DosSemSet( hsemKillThread );
  if(WinPostQueueMsg( hmqPrn, KILLTHREAD, MPFROMLONG(NULL), MPFROMLONG(NULL)))
      while( DosSemWait( hsemKillThread, (ULONG)-1));
  DosFreeSeg( selStack );
}
  :
  :
/*************************************************************************/
/* PRINTTHREAD:                                                          */
/* This thread carries out all processing associated with printing the   */
/* file that is currently being edited.                                  */
/*************************************************************************/
VOID far PrintThread()
{
QMSG     qmsgPrn;
BOOL     success;
```

```
habPrn = WinInitialize( (USHORT)NULL );
hmqPrn = WinCreateMsgQueue( habPrn, (SHORT)NULL );
/*****************************************************************************/
/* Set thread execution priority lower than that of the main thread.         */
/*****************************************************************************/
DosSetPrty( PRTYS_THREAD, PRTYC_NOCHANGE, -1, (TID)NULL );
if( hmqPrn != (HMQ)NULL )
     {
     /*****************************************************************************/
     /* Main message processing loop.                                             */
     /* Terminates on receipt of a KILL_THREAD message from the main              */
     /* thread.                                                                   */
     /*****************************************************************************/
     while (WinGetMsg( habPrn, &qmsgPrn, (HWND)NULL, 0, 0 ) )
        {
         if( qmsgPrn.msg == PRINTDOC )
             {
             /*****************************************************************/
             /* Simple flag used to show print in progress. It is             */
             /* maintained by this thread, and just monitored by the          */
             /* primary thread. Also used in the StatusProc window            */
             /* procedure to display when printing is in progress.            */
             /*****************************************************************/
             fPrinting = TRUE;
             success = PrintOut();          /* Call the printing function  */
             if (!success)
             WinMessageBox (HWND_DESKTOP, NULL, /* Display a message box  */
                            "An error has occurred!",     /* Message text  */
                            "Warning",1,MB_OK);           /* Title & style */
             fPrinting = FALSE;
             }
         }
     }
WinDestroyMsgQueue( hmqPrn );
WinTerminate( habPrn );
DosSemClear( hsemKillThread );
DosExit( EXIT_THREAD, 0 );
}
```

Example Code: Printing Using Device Calls

This code segment is taken from the AVIOSAMP.C sample program. It shows the use of the DevOpenDC (WDOPEN) and DevEscape (WDESC) calls. Changes that have been introduced to the sample program to demonstrate the use of message boxes are shown in **bold** print.

```
HAB habPrn;             /* print thread anchor block handle          */
/***************************************************************************/
/* Set default values in DEVOPENSTRUC                                      */
/***************************************************************************/
DEVOPENSTRUC  DevData  = {"LPT1Q",           /* address                  */
                         (PSZ)"IBM4201",     /* drivername               */
                         0L,                 /* driverdata               */
                         (PSZ)"PM_Q_STD",    /* datatype                 */
                         0L,
                         0L,
                         0L,
                         0L,
                         0L};
BOOL PrintOut()
{
  :
USHORT usJobId;
USHORT usLen;
PBYTE  pPrintBuf;
HDC    hdcPrn;
LONG   lOutCount = 0L;
BOOL   reply = FALSE;
  :
/***************************************************************************/
/* Create a device context for queued printer output.                      */
/***************************************************************************/
hdcPrn = DevOpenDC( habPrn,
                    OD_QUEUED,              /* type                     */
                    (PSZ)"*",               /* default token            */
                    4L,                     /* count                    */
                    (PDEVOPENDATA)&DevData, /* pointer to data          */
                    (HDC)NULL );            /* comp                     */

if( hdcPrn != (HDC) NULL )
    {
    /***********************************************************************/
    /* Indicate that a new print job is starting.                          */
    /***********************************************************************/
    if( DevEscape( hdcPrn,
                   DEVESC_STARTDOC,
                   7L,
                   (PBYTE)"AvioDoc",        /* document name            */
                   (LONG)0,
                   (PBYTE)NULL ) )
       /********************************************************************/
       /* Send data from the buffer to the printer                         */
       /********************************************************************/
       if( DevEscape( hdcPrn,
                      DEVESC_RAWDATA,
                      (LONG)usLen,
```

```
                    (PBYTE)pPrintBuf,
                    (LONG)0,
                    (PBYTE)NULL ))
            {
            /*************************************************************/
            /* Indicate the end of the print job                         */
            /*************************************************************/
            lOutCount = 2;
            DevEscape( hdcPrn,
                    DEVESC_ENDDOC,
                    (LONG)0,
                    (PBYTE)NULL,
                    (LONG)0,
                    (PBYTE)&usJobId );
            reply = TRUE;
            }
          else
            /*************************************************************/
            /* Cancel the print job if an error is detected              */
            /*************************************************************/
            DevEscape( hdcPrn,
                    DEVESC_ABORTDOC,
                    (LONG)0,
                    (PBYTE)NULL,
                    (LONG)0,
                    (PBYTE)NULL );
        DevCloseDC( hdcPrn);
        }
    DosFreeSeg( SELECTOROF( pPrintBuf ) );
    return( reply );
}
```

Chapter 37. Creating a Printer Device Context

A device context is a logical description of a physical device. Before you can send data to a printer using the device function calls, you must define a device context for it. You use the DevOpenDC (WDOPEN) call to do this. This chapter describes how to get the data that you must supply as parameters to this call, and how to use the call.

There are four items that you must allow the user to specify when your application creates a print job:

- The particular printer that will print the job
- The name of the presentation driver that controls the printer
- The name of a queue that is associated with the printer (if you want to use the spooler)
- Print options, such as print quality, form name, and page orientation.

When one of these items is set, the choice of the other three is usually limited. For example, if the user wants to use a particular printer, there may be only a limited number of presentation drivers that support it. Your application can display a list of these drivers. If the spooler is selected, you should also allow the user to specify which queue to use. There may be more than one printer serving this queue. In this case you should allow the user to select a printer. You could also present a list of available forms.

Your application must pass these items as parameters to the DevOpenDC (WDOPEN) call when it creates a printer device context.

The DevOpenDC (WDOPEN) Call

The parameters to the DevOpenDC (WDOPEN) call are described below, and the call is described in the *Programming Reference: Volume 1*. A typical use of the DevOpenDC call is shown in "Example Code: Creating a Printer Device Context" on page 37-10.

The parameter returned when you use the DevOpenDC (WDOPEN) call to create a device context is the device-context handle. This is an identifier for the device context. When you are printing using the device function calls, you use the device-context handle to address instructions to the correct device context. When you are printing graphic data, you also use it with the GpiAssociate (GSASS) call to associate a presentation space with the device context.

The parameters are:

hab The anchor-block handle.

type The type of device context. There are four types relevant to printing:

1. Use a *queued* (OD_QUEUED) device context to send output data to the spooler.
2. Use a *direct* (OD_DIRECT) device context to send output data direct to the printer.
3. Use an *information* (OD_INFO) device context when you want to retrieve information from a device, and not produce any output.
4. Use a *memory* (OD_MEMORY) device context to store a bit map.

token The device-information token. Always use an asterisk (*) for this parameter. This forces the system to get device information from the **data** parameter (see "Example Code: Creating a Printer Device Context" on page 37-10).

count The number of data elements supplied in the **data** parameter. This can be less than the full list of available elements, but must include null elements that lie between other data elements.

In the example code shown in "Example Code: Creating a Printer Device Context" on page 37-10, the driverdata element of the DevData structure is null, so to ensure the inclusion of the datatype element that follows it, the value of the **count** parameter of the DevOpenDC call must be 4.

In the case of a memory device context, the count parameter can take the value zero.

data The device-context data area. This is a pointer to a structure of the data type DEVOPENDATA. The elements of this structure are described in detail in "The Elements of the DEVOPENDATA Structure."

In the case of a memory device context, the data parameter can be null.

comp The compatible-device-context handle. This is used only when the **type** parameter is set to OD_MEMORY. It is the handle of the device context with which you want compatibility. An example of its use is given in "Printing a Copy of the Screen" on page 36-10.

The Elements of the DEVOPENDATA Structure

The DEVOPENDATA structure contains all the data needed to define the device context you want to open. It is recommended that your application uses the query calls described in "Selecting the Output Device" on page 37-3 and "Getting Form Selections and Job Properties" on page 37-4 to allow the user to define this data. The individual elements of the structure are described below. Note that the number of elements you need depends on the type of device context you are creating:

- For a *queued* device context you must use at least the address, drivername, datatype, and queueprocname elements.
- For a *direct* device context, the only relevant elements are address, drivername, driverdata, and datatype.
- For an *information* device context, you need at least address and drivername.
- For a *memory* device context, no elements of the DEVOPENDATA structure are used because the **comp** parameter in DevOpenDC gives sufficient information.

address
For a *queued* device context, this is the name of the queue for the device, for example, LPT1Q.

For a *direct* device context, this is the logical address of the device (the name of the port to which it is attached), for example, LPT1.

For an *information* device context, this is the logical address of the device that you want information about, for example, LPT1. See "Selecting the Output Device" on page 37-3 for information on how to query what printers and queues are available on the system.

drivername
This is a character string identifying the presentation driver, for example, IBM4201. See "Selecting the Output Device" on page 37-3 for information on how to query what presentation drivers available on the system.

driverdata
If the presentation driver allows the application to set options, they can be specified in the DRIVDATA structure. For example, the IBM presentation drivers allow the user to set options such as print quality and page orientation.

To use the default settings from the system initialization file, specify a null **driverdata** element.

The DRIVDATA structure is also used when one presentation driver supports more than one device. This is the case with the PLOTTERS.DRV driver supplied with Presentation Manager. To use a particular plotter, specify the name of the driver (PLOTTERS) in the **drivername** element of the DEVOPENDATA structure. Then give the name of the plotter (for example, IBM6180) in the **devicename** element of the DRIVDATA structure. This can be done in the C language with this code:

```
DEVOPENDATA dop;

   DRIVDATA   drivdata;
   drivdata.cb       = 44L;
   drivdata.lVersion = 0L;
   strcpy(drivdata.szDeviceName, "IBM6180");

dop.pszLogAddress   = "LPT1Q";
dop.pszDriverName   = "PLOTTERS";
dop.pdriv           = &drivdata;
dop.pszDataType     = "PM_Q_STD";
```

The DRIVDATA structure is described in the *Programming Reference: Volume 1.*

datatype
Specify one of the standard spool-file formats (PM_Q_STD or PM_Q_RAW).

comment
For queued output, this is an optional character string that the Print Manager can display to the user. Use it to describe your application. The Print Manager displays only the first seven characters.

If you want the Print Manager to display a unique name for each print job, pass the job name when you call DevEscape (WDESC) with the DEVESC_STARTDOC code at the start of each job (see "Using the Device Function Calls" on page 36-6).

The comment string is ignored for direct output.

queueprocname
The Presentation Manager-supplied spooler queue processor is called PMPRINT. If you want to use a different queue processor, pass its name in the queueprocname element. See "Selecting the Queue Processor" on page 37-4 for information on how to query what queue processors are available on the system.

queueprocparams
The queue-processor parameters can include information such as the number of copies you want to print, and the size of the output area on the printed page. For a full description of the PMPRINT parameters, see "Passing Parameters to the PMPRINT Queue Processor" on page 37-7.

The queue-processor parameters are ignored for direct output.

spoolerparams
The Print Manager uses these parameters when it schedules the print jobs. The parameters are:

- The form names that identify the paper to be used, for example, `FORM=A4,A5,ENV`. You must include all the forms required by the print job. The form names must all be valid for the printer.
- The priority of the print job, for example, `PRTY=60`. You can use any integer in the range 1 through 99. 99 is the highest priority. The default priority value is 50.

The spooler parameters are ignored for direct output.

networkparams
For queued output, this is an optional parameter string that is used to specify network options, for example, `USER=ADMIN`.

The DEVOPENDATA structure is described in the *Programming Reference: Volume 1.*

Selecting the Output Device

To allow the user to select a printer from those that are available on the system, your application needs to get information about the printers, presentation drivers, and queues. To get this information you query the system initialization file, OS2SYS.INI.

Data in OS2SYS.INI is stored in a two-level hierarchy:

- The primary subdivision is the *application name*. This is a text string that is used to identify an application. The "application" in this context is the Presentation Manager.
- Each application name has one or more *keynames*, each of which identifies one item of information.

Your application can read this data using the PrfQueryProfileString (PRQPST) call. Pass the application name and keyname as parameters to the call in the following way:

1. To get the name of the default printer, use an application name of `PM_SPOOLER` and a keyname of `PRINTER`.
2. To get a list of all the printers that are available, use an application name of `PM_SPOOLER_PRINTER` and a null keyname.
3. To get information about a particular printer, use an application name of `PM_SPOOLER_PRINTER` and use the printer name (which you got from either step 1 or step 2) as the keyname. The string that is returned has the form:

 `<Addr>;<DDL>;<QL>;<NetOpt>`

 where:

 Addr is the logical address of the printer, for example, `LPT1`.

 DDL is a list of one or more presentation drivers. If there is more than one, they are separated by commas, for example, `IBM4201,IBM4202`. The default driver is listed first.

If the presentation driver supports more than one device, the elements of the list are of the form:

<drivername>.<devicename>

The PLOTTERS.DRV driver supplied with Presentation Manager supports more than one plotter. An example of the presentation-driver list that is returned for this driver is PLOTTERS.IBM6180,PLOTTERS.IBM6182;.

QL is a list of one or more queues. If there is more than one, they are separated by commas, for example, LPT1Q, LPT3Q. The default queue is listed first.

NetOpt is a string describing network options, for example, USER=ADMIN.

All the data described above is returned to your application as a character string. You need to parse the string to get the data you need for input to the DevOpenDC (WDOPEN) call when you create a device context for the chosen output device.

You can see an example of the use of the PrfQueryProfileString (PRQPST) call in the code segment shown in "Example Code: Creating a Printer Device Context" on page 37-10. The code segment also shows how to parse the string that is returned by the call so that the printer address and presentation-driver name can be assigned to variables.

Selecting the Queue Processor

Your application should query the initialization file to get the name of the queue processor. The procedure is similar to that described for printers in "Selecting the Output Device" on page 37-3. Use the PrfQueryProfileString (PRQPST) call with an application name of PM_SPOOLER and a keyname of QUEUE to get the name of the default queue processor.

Getting Form Selections and Job Properties

A printer can often handle more than one size or type of paper, and each is identified to the spooler by a *form name*. You can pass form names to the spooler as **driverdata** in the DEVOPENDATA structure of the DevOpenDC (WDOPEN) call when you create a printer device context. The presentation driver uses this information to set the **forms** parameter in the **spoolerparams** of DEVOPENDATA.

Note: If you do not specify form names in **driverdata**, you must ensure that your application puts the names in **spoolerparams**. The spooler uses **spoolerparams** to schedule print jobs.

You can pass *job properties*, such as print quality and paper orientation, to the spooler as **driverdata**.

Use the DevPostDeviceModes (WDPDM) call to collect the form selections and job properties. You can read them direct from the initialization file or you can instruct the presentation driver to display a dialog box that allows the user to make selections.

You need to call DevPostDeviceModes twice, first to find out how much storage is needed to hold the data, and second to collect the data. The first time you call DevPostDeviceModes, give the presentation-driver name, the device name, and the printer name as input parameters, together with a null pointer for the **driverdata** parameter. For example:

```
LONG bytes;
bytes = DevPostDeviceModes (hab,
              NULL        /* driverdata   */
              "PLOTTERS", /* drivername   */
              "IBM6180",  /* devicename   */
              "PLOTTER1", /* printer name */
              DPDM_POSTJOBPROP);
                     /* option, not used */
```

You can use the PrfQueryProfileString (PRQPST) call to get the driver name and device name for a given printer name. (The printer name is the name by which the device is known to the Control Panel). For example, if the device is known as PLOTTER1, pass this as the keyname and PM_SPOOLER_PRINTER

as the application name. An example of the string that may be returned from the initialization file is:

```
LPT2;PLOTTERS.IBM6180;LPT2Q;;
```

For more information on the PrfQueryProfileString (PRQPST) call, see "Selecting the Output Device" on page 37-3.

Allocate sufficient storage to hold the data returned by DevPostDeviceModes (WDPDM), then call DevPostDeviceModes a second time. This time, the **driverdata** parameter should contain a pointer to the allocated storage, for example:

```
DRIVDATA drivdata;
DevPostDeviceModes (hab,
              drivdata,   /* driverdata   */
              "PLOTTERS", /* drivername   */
              "IBM6180",  /* devicename   */
              "PLOTTER1", /* printer name */
              DPDM_POSTJOBPROP);
                /* Display a dialog box   */
```

The DPDM_POSTJOBPROP option instructs the presentation driver to display a dialog box that contains a list of all the forms that it supports, and that may also contain job properties. For example, the IBM presentation drivers allow the user to set the print quality and the page orientation. The default settings for the dialog box are read from the initialization file. If there is no data in the initialization file, the driver supplies its defaults. However, you can set the initial settings of the dialog box by passing them as **driverdata** in the DevPostDeviceModes (WDPDM) call. To do this, you must specify a null printer name.

When the user has changed or accepted the settings in the dialog box, the information is passed back to your application in a DRIVDATA structure.

Use the DPDM_QUERYJOBPROP option on the DevPostDeviceModes call if you want the presentation driver to return the current job properties from the initialization file without giving the user any options. If there is no data in the initialization file, the driver returns its defaults.

Querying the Presentation Driver

If you want to give the user more information about a form, you can get it from the presentation driver using the DevQueryHardcopyCaps (WDQHC) call. This returns the following information for each form that is supported by the printer:

- The ASCIIZ name of the form
- The width and the height of the paper in millimeters
- The clipping limits of the paper in millimeters
- The number of pels between clipping limits
- Whether this form is the one currently installed on the device, or if it is selectable from another paper bin.

You must create an information device context (using the DevOpenDC (WDOPEN) call) for the printer before you can use DevQueryHardcopyCaps (WDQHC) to get this information. The following example code shows how to create an information device context and how to use the DevQueryHardcopyCaps call.

Example Code: The DevQueryHardcopyCaps (WDQHC) Call

This example code shows how to use an information device context.

```
HDC      hdc;                                  /* Device-context handle */
LONG     no_of_forms;                          /* Number of forms       */
                                               /* the device            */
PHCINFO  info;                                 /* Forms-information     */
                                               /* structure             */

/*************************************************************************/
/* Open the information device context                                   */
/*************************************************************************/
DEVOPENDATA DevData = {"LPT1",
                       "IBM4201"};

hdc = DevOpenDC ( (hab)0,
        OD_INFO,                               /* type                  */
        "*",                                   /* default token         */
        2L,                                    /* count                 */
        &DevData,                              /* pointer to data       */
        (hdc)0);                               /* comp                  */

/*************************************************************************/
/* Query number of forms available on the device                         */
/*************************************************************************/
no_of_forms = DevQueryHardcopyCaps
                    (hdc,
                     0L,               /* Start at beginning of list */
                     0L,               /* Get number of forms        */
                     NULL);

/*************************************************************************/
/* Get forms data                                                        */
/*************************************************************************/
DevQueryHardcopyCaps (hdc,
                      0L,              /* Start with first form      */
                      no_of_forms,     /* Query all forms            */
                      info);           /* Structure to hold returned */
                                       /* data                       */

/*************************************************************************/
/* Close the information device context                                  */
/*************************************************************************/
DevCloseDC (hdc);                      /* Close the information      */
                                       /* device context             */
```

To get information on a specific form, pass the DRIVDATA structure (that you got from the DevPostDeviceModes call) in the DEVOPENDATA structure when you open the information device context. DevQueryHardcopyCaps (WDQHC) returns information on the form that is flagged as current.

To get general information on the capabilities of a device (such as color table and font support), use the DevQueryCaps (WDQCAP) call. DevQueryCaps use the same information device context as DevQueryHardcopyCaps.

Passing Parameters to the PMPRINT Queue Processor

The way that the queue processor processes a print job is controlled by the optional queue-processor parameters that you can specify in the **data** parameter of the DevOpenDC (WDOPEN) call (see **queueprocparams** on page 37-3). The Print Manager adds these parameter values to the spool file.

The PMPRINT queue-processor parameters allow you to:

- Specify the number of copies of a print job
- Restrict printing to a specific area on the page
- Specify which part of a picture is to be printed
- Specify color output, if your printer allows this
- Specify the foreground and background colors in a monochrome print.

Note: You do not need to use queue-processor parameters if all you want is a single copy of a picture, scaled to fit the page. These are the default settings of the queue-processor parameters.

The first parameter (COP) is used for all spool-file formats. The remaining parameters are valid for PM_Q_STD spool files only. Because PM_Q_STD data are used mainly for *graphic* data, these parameters are described in relation to the printing of picture files.

The PMPRINT queue-processor parameters are:

COP=*n*

The COP parameter specifies the number of copies of the spool file that you want printed. The value of *n* must be an integer in the range 1 through 999.

The default is `COP=1`.

ARE=C | *w,h,l,t*

The ARE parameter determines the size and position of the output area. This is the area of the physical page to which printing is restricted.

A value of `ARE=C` means that the output area is the whole page. Note, however, that the printer can never print outside its own device clip limits.

To size and position the output area at a specific point on the page, use `ARE=`*w,h,l,t*, where:

w, h are the width and height of the desired output area.

l, t are the offsets of the upper-left corner of the output area from the left (l) and from the top (t) of the maximum output area.

These four values must be given as percentages of the maximum output dimensions. The maximum output area is the area within the device clip limits.

Examples of the use of the ARE parameter are given in "Examples Using the ARE and FIT Parameters" on page 37-8.

The default is `ARE=C`.

FIT=S | *l,t*

The FIT parameter determines which part of the picture is to be printed. You can ask for the whole of the picture, scaled to fit the output area, or you can position the picture (real size) anywhere within the output area. This may mean that the picture is clipped at the boundaries of the output area.

A value of `FIT=S` causes the output to be scaled until the larger of the height or the width just fits within the defined output area. The aspect ratio of the picture is maintained.

To print the picture *real size*, use `FIT=`*l,t*, where *l,t* are the coordinates of the point in the picture that you want positioned at the center of the output area. *l* is measured from the left edge of the picture, and *t* is measured from the top edge. The coordinates must be given as percentages of the real dimensions of the picture.

Examples of the use of the FIT parameter are given in "Examples Using the ARE and FIT Parameters" on page 37-8.

The default is `FIT=S`.

XFM=0 | 1

The XFM parameter allows you to override the picture-positioning and clipping instructions that are given by the ARE and FIT parameters, including their defaults.

A value of `XFM=0` gives output as specified in the picture file. For example, applications that use many different forms may define different positions on each form for their output.

A value of `XFM=1` allows the appearance of the output to be determined by the settings of the ARE and FIT parameters.

The default is XFM=1.

COL=M | C

The COL parameter allows you to specify color output if you have a color printer.

A value of COL=M creates monochrome output (black foreground with no background color). This is supported by all devices.

A value of COL=C creates colored output. If you request colored output on a monochrome device, the presentation driver tries to meet your request. This can cause problems as the only color available is black. If the picture file specifies a red line on a blue background, both are drawn in black!

The default is COL=M when you are addressing a monochrome printer, and COL=C when you are addressing a color printer.

MAP=N | A

The MAP parameter allows you to decide how the neutral colors (those that are not specified in the picture file) are printed.

A value of MAP=N gives a *normal* representation of a screen picture on the printed page. This means that the page background is white and the foreground is black.

A value of MAP=A gives the reverse of the normal representation. This means that, on the printed page, the background is black and the foreground is white.

The default is MAP=N.

Notes:

1. Pictures from a spool file may be printed differently to those printed when the spooler is not selected. This is because the default setting without the spooler is to print pictures real size, whereas the default for the queue-processor FIT parameter is "scaled to fit" the output area.
2. The queue-processor parameters must be separated by one or more blank spaces, for example:

   ```
   COP=3 ARE=C FIT=S
   ```

 They can be listed in any order. Default values are used for parameters that are omitted or entered incorrectly.
3. Image data (data constructed by setting pels on or off) is not affected by the ARE and FIT parameters. Consequently, if your picture contains image data, the printed version will be different to the screen version if you use these parameters.

Examples Using the ARE and FIT Parameters

The ARE and FIT parameters are the parameters that you use to determine which part of your picture file is printed, and where on the physical page it is printed. Figure 37-1 on page 37-9 shows some examples of practical uses of these parameters.

Example 1 shows the output when both the ARE and FIT parameters are set to their default values. The result is a print of the whole picture, scaled to fit the full page.

Examples 2 and 3 show how to restrict the output area to a part of the page. Because the FIT parameter is left at its default value, the whole picture is scaled to fit the restricted output area.

Example 4 shows part of the picture printed real size, using the whole page as the output area.

Examples 5 and 6 show a part of the picture printed real size in different places on the page.

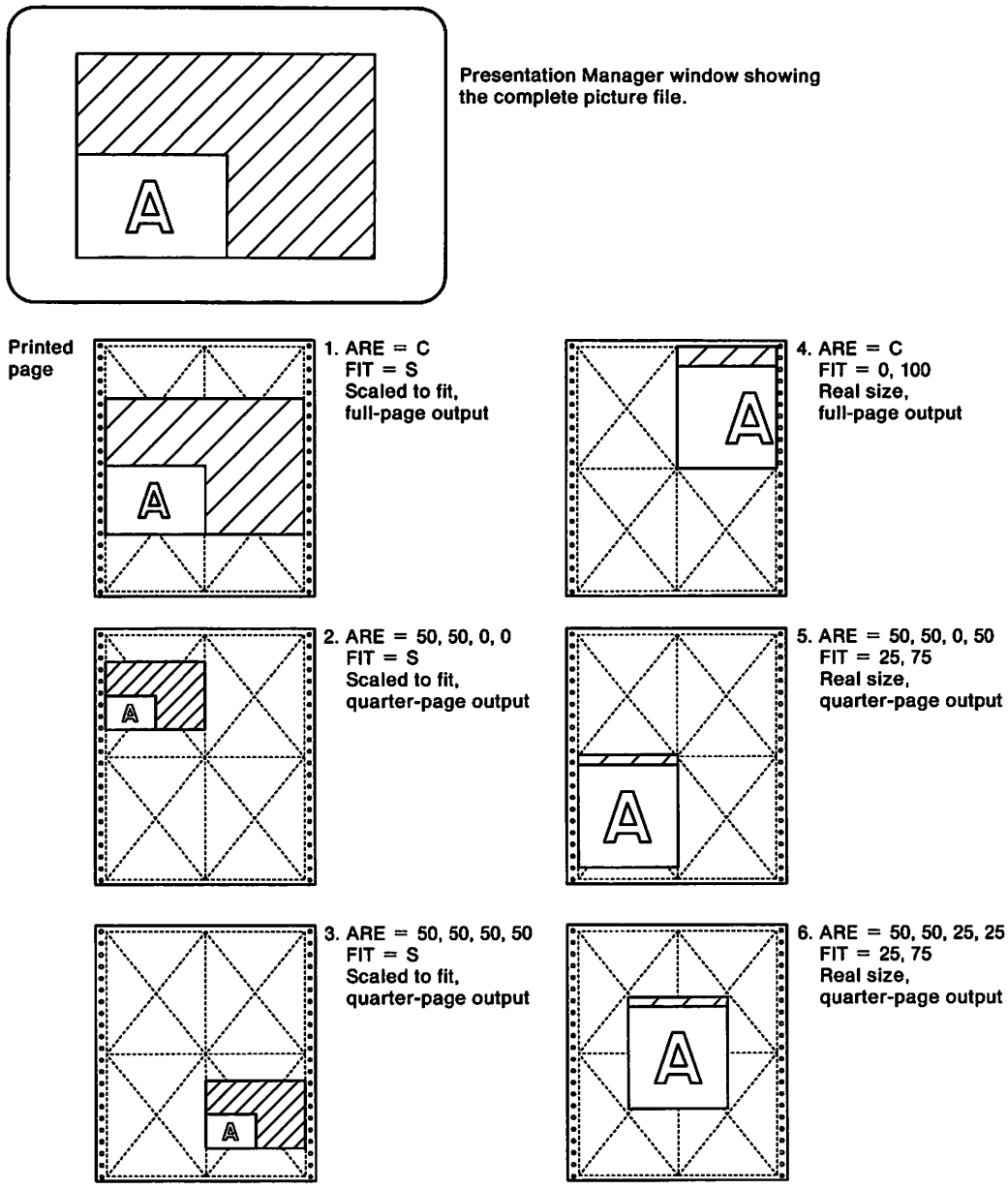


Figure 37-1. Examples Using the ARE and FIT Parameters

Example Code: Creating a Printer Device Context

The following code segment is taken from the AVIOSAMP.C sample program. It demonstrates the use of the PrfQueryProfileString (PRQPST) and DevOpenDC (WDOPEN) calls.

```
HAB habPrn;             /* print thread anchor block handle          */
USHORT usPsWidth;       /* presentation-space width,                 */
                        /* in AVIO units                             */
  :
/*****************************************************************/
/* Set default values in DEVOPENSTRUC                            */
/*****************************************************************/
DEVOPENSTRUC  DevData  = {"LPT1Q",          /* address               */
                         (PSZ)"IBM4201",    /* drivername            */
                         0L,                /* driverdata            */
                         (PSZ)"PM_Q_STD",   /* datatype              */
                         0L,
                         0L,
                         0L,
                         0L,
                         0L};
BOOL PrintOut()
{
USHORT usSize;
CHAR   szWrk[256];
CHAR   szTemp[PSWIDTH + 1];
HDC    hdcPrn;
PCH    pchScan;
  :
/*****************************************************************/
/* Get information about the default printer from OS2.INI and set up  */
/* the device data used in the DevOpenDC call.                   */
/* If reading OS2.INI fails, use default values that are in      */
/* DEVOPENSTRUC.                                                 */
/*****************************************************************/
if ( usSize = (SHORT)PrfQueryProfileString(HINI_PROFILE,
                     (PSZ)"PM_SPOOLER", /* application name          */
                     (PSZ)"PRINTER",    /* keyname                   */
                     NULL,                        /* default token    */
                     (PSZ)szTemp,       /* Name of default printer,  */
                                        /* returned from .ini file.  */
                     (LONG)sizeof(szTemp) ))  /* Maximum number of   */
                                              /* characters returned */
   {
   szTemp[usSize - 2] = 0;           /* remove terminating semi-colon */

   if (PrfQueryProfileString(HINI_PROFILE,
                             (PSZ)"PM_SPOOLER_PRINTER",
                             (PSZ)szTemp,
                             NULL,
                             (PSZ)szWrk,
                             (LONG)sizeof( szWrk ) ))
      {
      pchScan = strchr(szWrk, ';');
      pchScan++;
      pchScan = strchr(pchScan, ';');
```

```
        pchScan++;
        *(pchScan + strcspn(pchScan, ".,;")) = 0;
        DevData.pszLogAddress =  pchScan;           /* Get address     */

        pchScan = strchr(szWrk, ';');
        pchScan++;
        *(pchScan + strcspn(pchScan, ".,;")) = 0;
        DevData.pszDriverName = pchScan;            /* Get drivername  */
        }
    }
/*********************************************************************/
/* Create a device context for queued printer output.                */
/*********************************************************************/
hdcPrn = DevOpenDC( habPrn,
                    OD_QUEUED,              /* type                  */
                    (PSZ)"*",               /* default token         */
                    4L,                     /* count                 */
                    (PDEVOPENDATA)&DevData, /* pointer to data       */
                    (HDC)NULL );            /* comp                  */
  .
  .
  .
}
```

Chapter 38. Device Support

Device Drivers

A device driver is the interface between the operating system, together with its applications, and a physical device such as the keyboard, mouse, or printer. The device driver sends data to and receives data from a device, resolving device-independent requests from applications with the device-specific attributes of the device.

The primary method of communication between the operating system and a device driver is request packets. OS/2 receives input/output (I/O) requests from applications and sends data in the form of request packets to the device driver. The device driver communicates with the device through, for example, the BIOS and ABIOS interfaces.

The OS/2 1.2 Programming Tools do not provide sample programs for all function calls discussed in this chapter. However, when an example of a function call is used in a sample program, the subdirectory path of the sample program is shown. **Note that a sample program may not illustrate the task of the function call as discussed in the chapter text.**

Application I/O to Devices

An application makes I/O requests to devices through one of the following interfaces:

- File system interface
- Subsystem interface
- IOCtl interface
- Character device monitors
- IOPL code segments.

The File System Interface

The file system interface consists of the file I/O function calls. File I/O function calls are used primarily to perform I/O on block devices such as fixed disks. Some file I/O function calls can be used to perform I/O on character devices. These calls include:

DosOpen	**DosClose**
DosRead	**DosReadAsync**
DosWrite	**DosWriteAsync.**

The advantage of using the file I/O function calls to perform I/O on a character device is that the application can redirect the I/O. The application performs I/O to a handle (of a file, pipe, or device), which it obtained from opening the named resource passed to it.

Sample Programs: Examples of the **DosOpen**, **DosClose**, and **DosWrite** calls are found in the following subdirectories:

```
\TOOLKT12\C\SAMPLES\PM\AVIOSAMP\AVIOSAMP.C
\TOOLKT12\C\SAMPLES\PM\IMAGE\IMAGE.C
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROCESS.ASM
```

An example of the **DosReadAsync** call is found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\PROCESSB\PROG2.ASM
```

The Subsystem Interface

A subsystem interface shields an application from having to manage device I/O. The file system is an example of a subsystem. The file system allows the application to view a file as a logical sequence of sectors, thus shielding the application from having to manage the physical locations of the data on the disk media. Other subsystem examples include the video, keyboard, and mouse subsystems.

The IOCtl Interface

The IOCtl interface is the mechanism an application or subsystem uses to send device-specific control commands to the device driver. The IOCtl mechanism is DosDevIOCtl for OS/2 applications and the Interrupt 21H IOCtl request for applications running in DOS mode. I/O commands can be sent to both block and character device drivers. The application or subsystem must first obtain the device handle by issuing a DosOpen for the device name.

Character Device Monitors

OS/2 provides a mechanism for applications or subsystems to monitor all characters passing through a character device driver. This mechanism allows one or more registered processes (referred to as character device monitors) to remove, insert, or modify data passing through a character device.

The device monitor mechanism is supported by the OS/2 monitor function calls and the Monitor Dispatcher Device Helper.

OS/2 monitor function calls consist of a set of five dynamic link routines through which a character device monitor intercepts and filters data passing through a device. These routines are:

DosMonOpen **DosMonClose**
DosMonReg **DosMonRead.**
DosMonWrite

Monitor Dispatcher Device Helper support comprises a set of five OS/2 device helper routines through which the device driver creates and maintains monitor chains for each of its data streams. The routines are:

dh_MonitorCreate
dh_Register
dh_DeRegister
dh_MonWrite
dh_MonFlush.

These routines enable a device driver to field monitor registration requests from a character device monitor application and send data to the registered monitors in its monitor chains.

Character device monitors can provide a variety of functions, serving, for example, as:

- A device-specific user extension of a device driver
- A dialog manager
- A redirector of data from one device to another.

OS/2 character device drivers that provide monitor support include the keyboard, mouse, and printer device drivers. The OS/2 character device monitors that comprise the OS/2 product, include the print spooler. The spooler registers a monitor for each printer device. The monitor takes data from the printer device data stream and stores the data in files on a fixed disk. The monitor sends spooler data files to the printer device for printing by returning the data to the data stream.

Sample Programs: Examples of the **DosMonOpen**, **DosMonClose**, **DosMonReg**, **DosMonRead**, and **DosMonWrite** calls can be found in the following subdirectory:

```
\TOOLKT12\MASM\SAMPLES\BSE\DEMODB\MONITOR.ASM
```

Device Monitors and Presentation Manager Sessions

Applications (including both OS/2 and Presentation Manager applications) may not monitor key strokes or mouse clicks for Presentation Manager sessions. The key stroke and mouse click mechanism for Presentation Manager sessions differs from the mechanism for OS/2 sessions; that is, sessions used by full-screen applications.

The keyboard and mouse device drivers manage key strokes and mouse clicks for each OS/2 session. These device drivers create monitor chains for an OS/2 session, and an application may register a key stroke or mouse monitor for any of these sessions.

The Presentation Manager manages both key strokes and mouse clicks for each of its extended sessions. The Presentation Manager does not create monitor chains for each of its extended sessions.

All applications (including OS/2 and Presentation Manager applications) may, however, monitor printer data. The printer device driver defines its data streams and monitor chains on a per physical device basis.

In general, data streams for character devices that are not defined on a per session basis may be monitored by a character device monitor, provided the device driver contains device monitor support.

For a more detailed description of character device monitors, see *I/O Subsystems and Device Support: Volume 1*.

IOPL Code Segments

When a standard device driver does not provide sufficient service to and from an application, an OS/2 application may directly access a device through service routines defined in its own IOPL code segments.

Permission to access a device is requested by calling:

DosPortAccess
DosCLIAccess.

Since a device driver is the standard way to provide service to and from applications, IOPL code segments should be used only when the standard device driver is not sufficient.

For more information on IOPL code segments, see *I/O Subsystems and Device Support: Volume 1.*

Presentation Drivers

Presentation drivers are special purpose I/O routines that field device-independent I/O requests from the Presentation Manager and its applications, resolving them with the device-specific attributes of the device. In this way, they are similar to OS/2 device drivers.

However, they are a higher-level interface for output devices than the OS/2 device drivers. OS/2 device drivers are loaded at the most privileged level, ring 0. Presentation drivers are loaded at privilege level 2. Both OS/2 device drivers and presentation drivers have I/O privilege.

Presentation drivers are also subsystems. They resolve I/O requests from the Presentation Manager and its applications by interfacing directly with the device or with its device driver. Printer and plotter presentation drivers use the spooler interface calls to pass data and controls on to the OS/2 kernel (that is, the device drivers). Screen presentation drivers pass data and controls directly to the display hardware interface.

Presentation drivers are supplied as dynamic link libraries. When the Presentation Manager is initialized, the screen presentation driver is automatically loaded and enabled. Other presentation drivers (for example, those for the printer and plotter) are loaded and enabled in response to an application calling the DevOpenDC (WDOPEN) function to create a device context.

For more information on presentation drivers, see *I/O Subsystems and Device Support: Volume 2.*

Sample Programs: Examples of programming techniques for coding a presentation driver are found in the following subdirectory:

```
\TOOLKT12\PD\SAMPLES\PMPRT
```

Part 9. Product Development Considerations

Chapter 39. Supporting National Languages

Many applications need to support more than one national language, for example, French and German. This chapter describes the facilities that OS/2 provides to support national languages.

The support of national languages by applications involves three main considerations, as follows:

1. Any text displayed by the program must be translated into the appropriate language.
2. Any symbols or icons used by the program may not convey the same meaning in all countries. Therefore, these should be considered carefully and, if necessary, alternative designs produced for different countries.
3. Different text characters are often needed by different languages.

The first two considerations are simplified through the use of resources.

The third consideration is assisted by the ability of the application to receive input in any ASCII code page and to display data in any ASCII code page. The ASCII code page 850 avoids many of the problems in this area, as it contains most of the characters needed for most languages. However, older programs use code page 437 for U.S. English and code pages 860, 863, and 865 for various other languages. The code page applies to both output and input data.

Note: Presentation Manager text was translated for non-English versions using code page 850. It is strongly recommended that code page 850 is used, whenever possible, for all PM applications that may require translation.

Resources and National Language Translation

The simple rule when constructing a Presentation Manager application is to define any national language dependencies in resources that are held in resource files separate from the code.

Therefore:

- Pure text strings should be held in string tables
- Menus should be held in menu templates
- Dialog boxes should be in dialog templates
- Accelerators should be in accelerator tables.

In most instances, you can change the language displayed by your application by translating the resources, without changing the application.

For dialog boxes and text strings, however, you may have to alter the application code to take account of the change in length of text strings when they are translated. For example, changing the length of a string can cause it to become misaligned relative to other displayed objects.

You can use the Dialog Box Editor to correct misalignments, or increase or decrease the overall size of the dialog box if necessary. This can save you from changing your application code.

Strings that are explicitly displayed by the application (such as using GpiCharString (GSCHAP)) are more of a problem. If you are designing an application to support different national languages, the best option is to include code that can cope with strings of varying length and format them according to their size.

Selecting a Language Version of an Application

There are two simple methods of obtaining a specific national language version of a program designed to cope with a range of languages. The choice between the two depends on factors such as the amount of disk space available and whether the user needs to change between languages once the program is installed:

1. Statically link the resources into the application's .EXE file(s). In this case, the executable files are language-specific and cannot be changed to another language. The specific .EXE file(s) can then be shipped to the user.
2. Place the resources into a language-specific, dynamically-linked library. There is one such file for each language. Selecting a particular file for use with the application gives a particular version of the program.

This technique can allow all the languages to be shipped, with selection of the language during installation (by, for example, appropriate naming of a specific .DLL file). It is possible for the language setting to be determined while the program is running, for example, the user can change between French and German while the program is running.

Translating Messages in Full-Screen Applications

For **full-screen applications**, text messages (used to display information to the user) are held in a *message file*. Create separate message files for each national language that you want to support. Use the MKMSGF utility to create a binary-format version of a message file, and the MSGBIND utility to bind the binary file to the application's executable file.

OS/2 provides a series of API calls to retrieve a message from a message file, and insert the message into the application:

- **DosGetMessage** retrieves a message from a specified message file and inserts variable text-string information into the body of the message.
- **DosInsMessage** also inserts variable text-string information into the body of a message, but does not retrieve the message. Use DosInsMessage when you want to load messages before you know the text of the strings that are to be inserted into them.
- **DosPutMessage** outputs a message, usually to a display screen. DosPutMessage formats the screen buffer to prevent words being split at the end of a line.

There are examples of the DosGetMessage and DosPutMessage calls in the following sample programs:

```
\TOOLKT12\C\SAMPLES\BSE\CRERRCB\CRERRC.C
\TOOLKT12\C\SAMPLES\BSE\MEMORYCB\MEMORYC.C
\TOOLKT12\C\SAMPLES\BSE\DLCBAT\DLNK3C.C
\TOOLKT12\C\SAMPLES\BSE\MOUSECB\MOUSEC.C
\TOOLKT12\C\SAMPLES\BSE\WTCBAT\WTC.C
\TOOLKT12\MASM\SAMPLES\BSE\MEMORYB\MEMORY.ASM
\TOOLKT12\MASM\SAMPLES\BSE\FAMAPI\VIOSAMP.ASM
\TOOLKT12\MASM\SAMPLES\BSE\CRERRB\CRERR.ASM
\TOOLKT12\MASM\SAMPLES\BSE\DYNBLD\DYNLINK.ASM
```

Note: Additional information on creating text message files using the MKMSGF and MSGBIND utilities are given in *Building Programs*.

Code Page Support

OS/2 code-page management reads keyboard input and writes display and printer output for concurrent multiple processes that input and output ASCII-based character data encoded in different code pages.

A code page defines a character set by assigning each character to a location in a table. A character set is implemented by using a *character shape* table from which a character is selected for output by its associated display or printer. A character set is either downloaded to a device or is ROM resident at the device.

The system does this by switching the required code page, for a code-page-supported device, before input or output.

The required code page is the current code page of the process, at the time it opens a device, or a specific code page selected by the process with a set code-page API function. A character set can also be plugged in the device, such as with the IBM Quietwriter printer.

In addition, the country APIs retrieve country- and language-dependent information in either the current code page of the calling process or in a code page selected by the process.

Code Page Management

Code Page Management allows a user to select a code page for keyboard input and screen and printer output before running an application, a system command, or utility, in the OS/2 multitasking environment.

This allows the user in a particular country, such as England (code page 437), Norway (code page 865), or a language region such as Canadian-French (code page 863), to run with a code page that defines an ASCII-based character set containing characters used by that particular country or language.

OS/2 supports the following code pages:

437 U.S. IBM PC code page
850 Multilingual code page
860 Portuguese code page
863 Canadian-French code page
865 Nordic code page.

Presentation Manager applications that need to use an EBCDIC code page, as an alternative to an ASCII code page, can set such a code page with the VioSetCp and GpiSetCp (GSCPG) calls. This code page can be one of the following Country Extended Code Pages (CECPs):

037 U.S. English
297 French
273 Austrian/German
274 Belgian
280 Italian
284 Spanish
285 U.K. English
277 Danish/Norwegian
278 Finnish/Swedish
282 Portuguese
500 International.

Each of these code pages is based on character set 697.

Any code page that is either defined in CONFIG.SYS, or is one of the EBCDIC code pages supported, may be selected. Installable code-page files include keyboard translate tables, display character sets, printer character sets, and country/language information for each code page supported.

The primary features of Code Page Management are:

- The statements (CODEPAGE and DEVINFO) in CONFIG.SYS, for commanding system initialization to prepare selected code pages and devices for code-page switching, at run time, when in character text modes.
- The user command CHCP, changes the code page of its session (command process that executes CHCP) and any application that runs in that session. CHCP affects printer code pages only on print jobs that are opened subsequent to the command.
- API functions for an application to set and query the code page for a process, keyboard input, video output, and spooled printer output.
- System code-page switching of keyboard, display, printer/spooler, and country information for concurrent processes in different code pages.

One system country code, one system keyboard layout, and up to two system code pages can be configured for system use at run time.

Code Page Preparation

During system initialization, the selected code pages specified in the CODEPAGE statement are prepared to allow run-time code-page switching of the display, the keyboard, the printer, and the country information. The display, keyboard, and printer to be prepared, must be defined in a DEVINFO statement. Country information is prepared for the system country code specified in the COUNTRY statement. If a resource cannot be prepared for the selected code page, it is prepared for a default code page.

System resources default at system initialization for code-page preparation when the code page cannot be found for the resource. The following are the defaults:

- A keyboard layout defaults to the code page of the translate table designated as the default layout in the KEYBOARD.DCP file. The default layout is based on the national code page of its associated country. KEYBOARD.DCP must be explicitly specified in the DEVINFO statement for the keyboard in CONFIG.SYS.
- The display defaults to the code page of ROM_0 for the device.
- The printer defaults to the code page of ROM_0 for the device.
- The country information defaults to the code page of the first entry found in the COUNTRY.SYS file for the country code. Each entry is the same information for a given country code but encoded in a different code page. The first entry is based on the preferred country code page.

Note: ROM_0 means a device default code page that is the device native code page or the lowest addressed ROM code page.

In the following table, shown left to right are the country code default code pages, and the country keyboard layouts. If country information cannot be prepared at system initialization because it is not found in the COUNTRY.SYS file, for a code page selected with the CODEPAGE statement, it is prepared (maintained for run-time code-page switching in memory) in the default code page. Similarly, a keyboard layout is prepared in its default code page if it cannot be prepared in the selected code page, because it is not found in the KEYBOARD.DCP file. COUNTRY.SYS contains one default entry per country code and KEYBOARD.DCP contains one default entry per keyboard layout based on these assignments:

Country	Ctry Code	Code Page	Kbd
Asia	099	437	-
Australia	061	437	-
Belgium	032	437	BE
Canada	002	863	CF
Denmark	045	865	DK
Finland	358	437	SU
France	033	437	FR
Germany	049	437	GR
Italy	039	437	IT
Latin America	003	437	LA
Netherlands	031	437	NL
Norway	047	865	NO
Portugal	351	860	PO
Spain	034	437	SP
Sweden	046	437	SV
Switzerland	041	437	SF,SG
United Kingdom	044	437	UK
United States	001	437	US

Although up to two code pages can be selected with the CODEPAGE statement, the system may actually have prepared three or more code pages at system initialization in case a system resource defaults on code-page preparation. For example, the keyboard may be prepared for a default code page that is different from the two selected code pages for which the display is successfully prepared.

Statements

Use the following statements in CONFIG.SYS to prepare and define a code page:

COUNTRY
Specifies the country-dependent information, such as currency and date/time format. Default: the first entry in the COUNTRY.SYS file (based on the preferred national language character set).

DEVINFO
Defines the display, keyboard, and printer that are to be prepared for code-page switching. Default: the translate table designated in the KEYBOARD.DCP file (for keyboard) and the device ROM_O (for the display and printer).

CODEPAGE
Specifies the code pages to be prepared to allow run-time code-page switching of the display, keyboard, printer, and country information. Default: the first entry in the COUNTRY.SYS file based on the preferred national language character set.

Note: The active code page can be changed at any time using the CHCP statement.

Function Calls

At the system level, OS/2 switches the code pages of supported displays and printers to agree with the code page of the process sending the output. At the application level, OS/2 function calls allow a process to control code page-assignments.

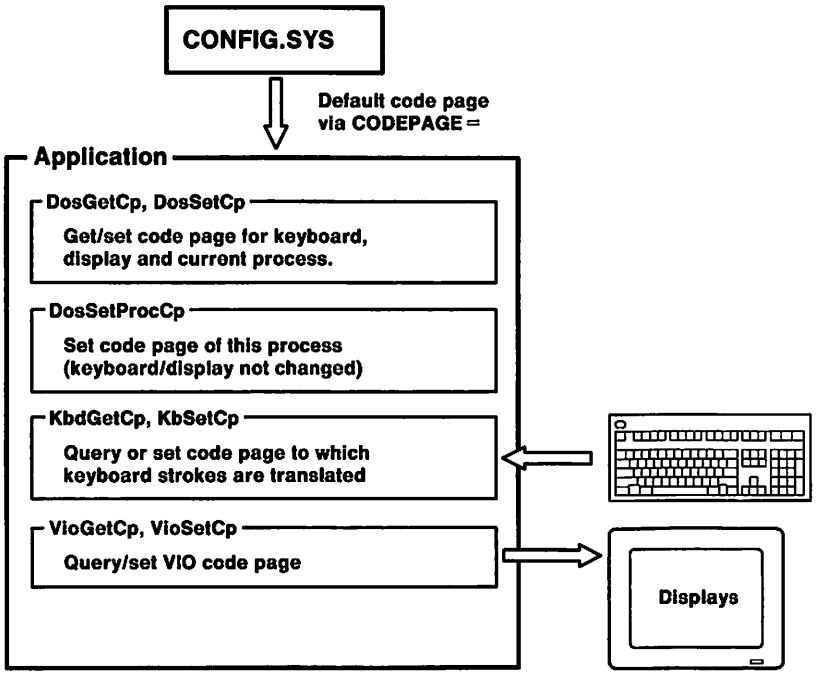


Figure 39-1. Code page options for VIO applications

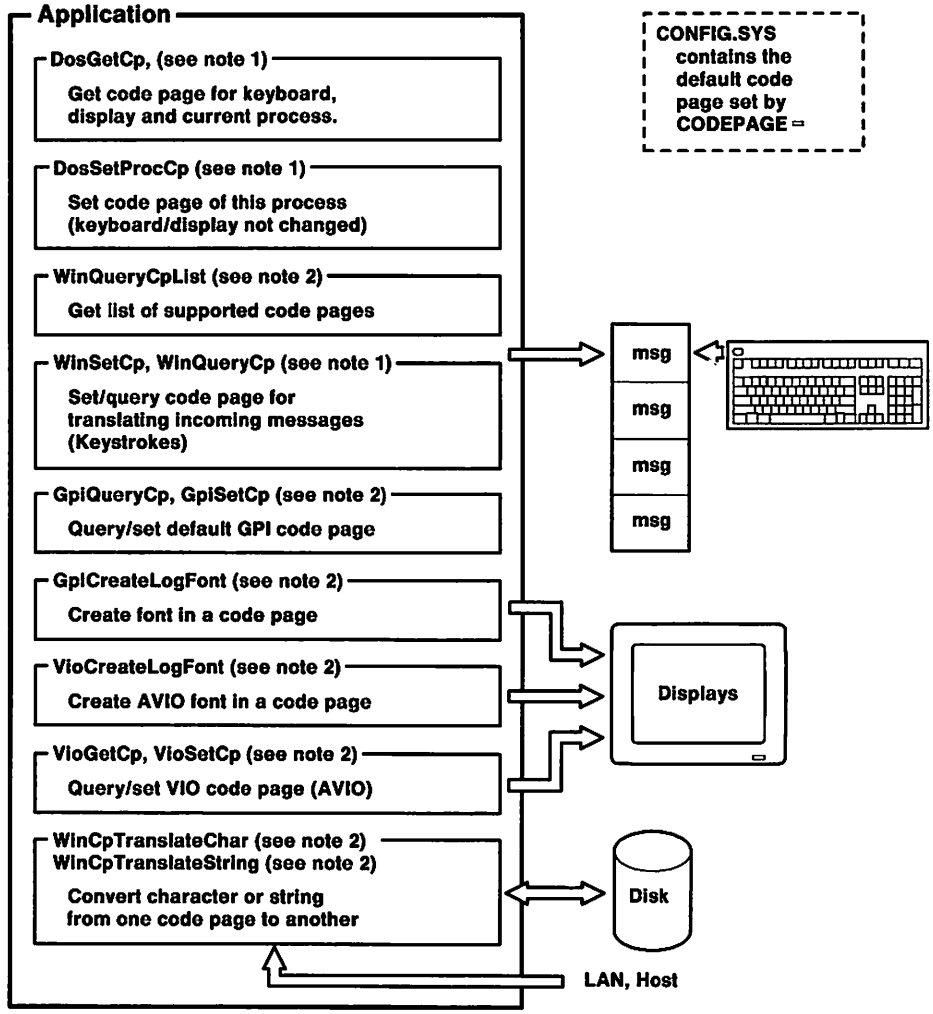
Code Page Operation

Each process has a code-page tag maintained by the OS/2 kernel. A code page is a table that defines how characters are encoded. A code-page tag is the identifier of the current code page for the process.

A child process inherits the code-page tag value of its parent. The default code page for the first process of a program started in a session is the same as the session code page. The default code page for a new session is the primary code page specified in the CODEPAGE configuration statement. A process code-page tag may be changed with the function call DosSetCp or DosSetProcCp and queried by DosGetCp, but, DosSetCp or DosSetProcCp does not change its parent or any child process code-page tags. The actions of these calls are illustrated in Figure 39-1. Further details are given in the *Control Program Programming Reference*.

The following paragraphs explain the uses of code pages when performing process input and output:

- Spooled printer output is printed by the spooler in the code page of the process. The code page the spooler uses to print is established through the system spooler and file system at the time the process makes an open printer request.
- Video output is shown by the Video Subsystem in the current code page of the implied video handle being used for output by the process. The default display code page for a new session is the primary code page specified in the CODEPAGE configuration statement or defaults to the display ROM code page. The display code page can be changed by a process with VioSetCp or DosSetCp for the logical display of the session to which the calling process belongs.
- Keyboard scan-code input is converted into ASCII characters by the keyboard device driver in the current code page of the keyboard handle being used for input by a process. The default keyboard handle code page for a session is the primary code page specified in the CODEPAGE configuration statement or defaults to US437. The keyboard code page can be changed by a process with KbdSetCp or DosSetCp for the logical keyboard of the session to which the calling process belongs.
- Utility and command output is shown in the current code page of the command process that executes the function.



Notes:

1. Either of the two ASCII code pages in CONFIG.SYS
2. Any supported ASCII or EBCDIC code page as reported by WinQueryCpList (WMQCPL)

Figure 39-2. Code page options for PM applications

Any invocation of a CMD.EXE process in OS/2 or a COMMAND.COM process in DOS is a session. Therefore, the code page of the command process is the session code page. The user command CHCP is provided to change the code page of a session. CHCP performs the following functions:

- Sets the session's command process code page
- Sets the session's logical display code page
- Sets the session's logical keyboard code page.

The printer code page is based on the process code page at the time the process makes an open printer request. No special command is needed for the printer.

Presentation Manager extends the code page options with an additional set of calls. The action of all the available calls is shown in Figure 39-2

Presentation Manager applications make use of additional calls to perform process input and output:

- The codepage for Presentation Manager is the first entry in the CODEPAGE= statement in CONFIG.SYS (as returned by DosGetCp).
- Keyboard scan-code input, from the keyboard message queue, is converted into ASCII characters by the keyboard device driver, using the current code page for the keyboard set by WinSetCp.. WinQueryCpList (WMQCPL) returns a list supported code pages, and WinQueryCp (WMGECP) gives the current code page.
- The code page for graphics characters is set by GpiSetCp (GSCPG), and may be queried by GpiQueryCp (GSQCPG). Additionally, a font, for use within the code page, can be created with the GpiCreateLogFont (GSCRLF) call.
- The ability to create a font is also allowed in Full-screen applications by using the VioCreateLogFont call.
- Individual characters, or complete strings of characters, can be converted from one code page to another by the WinCpTranslateChar (WUTRCH) and WinCpTranslateString (WUTRST) calls. This allows a Presentation Manager application to access data stored by another application, or to write data for another application, using a different code page.

Code-Page-Dependent Information

System information dependent on the code page includes:

- Video character set for display
- Keyboard translate table for scan code to character conversion
- Printer translate table and ROM code pages for printer
- Country format information for time, date, and other formats
- Language collate sequence table for character string sorting
- Language case map table from lowercase to uppercase
- Language DBCS environment vector of lead bytes.

This information is created at system initialization and is maintained in system storage by OS/2 for each system code page that is prepared. The information is retrieved as needed according to the code page.

Code-Page Switching

Applications written for a specified code page can have code page usability problems when executed with a different code page. Menu boxes, for example, can become strings of accented characters, or help text become unreadable because letters have changed to graphics. You can avoid these problems by using the appropriate hardware that supports multiple code pages through code-page switching.

Some devices such as the EGA and Proprinter, support downloadable fonts that can be switched to the appropriate code page before an application is run. Applications developed using different code pages will then run on the same IBM PC without software modification.

Code-page switching also allows the flexibility to develop new applications based on non-Latin based or special industry-based code pages.

Code-Page Switching Examples

Some examples of code-page switching that can occur are:

- The printer character set is switched by the Print Spooler to the process code page (which opens the printer) before it outputs the process write data stream to the printer.
- A process requests DosGetCtryInfo for the country information of the system country code or another selected country code. The information is provided encoded in the code page of the process.

Switching the display to the code page that a system ASCII message is encoded in, before output, is not provided.

File names created under one code page may not be accessible by that name under another code page as the name characters may case map differently. This can be avoided by using only the first 128 characters of a code page for file names or only unaccented characters (a-z, A-Z, and 0-9).

Character Set

A "character set" is a set of shapes, arranged in a table (code page), that represents the alphabet, numbers, and other special symbols related to a language. A typical code page has 256 positions in the table. Each character symbol is assigned an 8-bit binary code by its code page table.

- As languages fall into categories, it is possible for a character set to contain the shapes appropriate for several languages.
- Both the computer hardware and the software must know which code page is being used, in order to print, process, or display the correct characters.
- Just as there are different character sets for different languages, so there are also different code pages. The code pages allow your computer system to print, process, and display national language characters for many different languages.

A given code page can be displayed or printed with more than one font when the fonts differ in pitch, type style, or proportional spacing.

Code Page 437

Code Page 437 (CP437) (see Figure 39-3 on page 39-9) is the standard PC code page familiar to English-speaking PC users.

Note: Some of the 256 displayable symbols are printer control characters and do not print.

The lower 128 characters are based on the 7-bit ASCII code. The upper 128 characters contain characters from several European languages (including part of the Greek alphabet) and various graphic characters. However, some of the accented characters, such as those used in the Nordic countries, are not represented. The missing characters are available in other versions of the code pages.

Code Page 850

Code Page 850 (CP850)(see Figure 39-4 on page 39-9), also called the multilingual code page, allows effective communication and data exchange between IBM PCs and host mainframes.

CP850 supports the alphabetic characters of all Latin-based languages. It contains all the characters required by 13 languages used in approximately 40 countries.

Languages Supported by CP850

CP850 enables the transfer of Latin language-based data between countries using the following languages:

Belgian French	Canadian French
Danish	Dutch
Finnish	Flemish
French	German
Italian	Norwegian
Portuguese	Spanish
LAD Spanish	Swedish
Swiss French	Swiss German
U.K. English	U.S. English.

Differences Between CP437 and CP850

CP850 (Multilingual) is different from CP437 (Universal English) in the following ways:

- CP850 duplicates some characters below X'20' (the paragraph and section symbols, X'14' and X'15') to the region above X'80'. This change makes the characters easier to print on some printers.
- CP850 replaces the single-double box graphics with accented characters.
- CP850 replaces the Greek letters with accented characters.
- CP850 moves the cent and Yen symbols to other locations on the code page. The former location of the cent and Yen signs are replaced with a slashed 'o'. The Nordic characters are in the same position as they are in the Nordic variant of CP437.
- CP850 also replaces the square root symbol with a superscript "1".

Software developers can prepare products for operation with CP850 by using only the CP850/CP437 common code points in their executable code. CP850 provides the flexibility to develop new applications based on non-Latin based or special industry-based code pages.

Hex Digits 1st → 2nd ↓	0-	1-	2-	3-	4-	5-	6-	7-		8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p		Ç	É	á	▒	└	╨	α	≡
-1	☺	◄	!	1	A	Q	a	q		ü	æ	í	▒	┴	╤	β	±
-2	☻	↕	"	2	B	R	b	r		é	Æ	ó	▓	┬	╥	Γ	≥
-3	♥	!!	#	3	C	S	c	s		â	ô	ú	│	├	╙	π	≤
-4	♦	¶	$	4	D	T	d	t		ä	ö	ñ	┤	─	╘	Σ	⌠
-5	♣	§	%	5	E	U	e	u		à	ò	Ñ	Á	┼	╒	σ	⌡
-6	♠	▬	&	6	F	V	f	v		å	û	ª	╡	╞	╓	µ	÷
-7	•	↨	'	7	G	W	g	w		ç	ù	º	╢	╟	╫	τ	≈
-8	◘	↑	(	8	H	X	h	x		ê	ÿ	¿	╖	╚	╪	Φ	°
-9	○	↓	)	9	I	Y	i	y		ë	Ö	⌐	╕	╔	┘	Θ	∙
-A	◙	→	*	:	J	Z	j	z		è	Ü	¬	╣	╩	┌	Ω	•
-B	♂	←	+	;	K	[	k	{		ï	¢	½	╗	╦	■	δ	√
-C	♀	∟	,	<	L	\	l	\|		î	£	¼	╝	╠	▬	∞	n
-D	♪	↔	-	=	M	]	m	}		ì	¥	¡	╜	═	▌	ϕ	2
-E	♫	▲	.	>	N	^	n	~		Ä	Pt	«	╛	╬	▐	ε	■
-F	☼	▼	/	?	O	_	o	⌂		Å	ƒ	»	┐	╧	▬	∩	

Figure 39-3. Code Page 437

Hex Digits 1st → 2nd ↓	0-	1-	2-	3-	4-	5-	6-	7-		8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p		Ç	É	á	▒	└	ð	Ó	-
-1	☺	◄	!	1	A	Q	a	q		ü	æ	í	▒	┴	Đ	β	±
-2	☻	↕	"	2	B	R	b	r		é	Æ	ó	▓	┬	Ê	Ô	=
-3	♥	!!	#	3	C	S	c	s		â	ô	ú	│	├	Ë	Ò	¾
-4	♦	¶	$	4	D	T	d	t		ä	ö	ñ	┤	─	È	õ	¶
-5	♣	§	%	5	E	U	e	u		à	ò	Ñ	Á	┼	ı	Õ	§
-6	♠	▬	&	6	F	V	f	v		å	û	ª	Â	ã	Í	µ	÷
-7	•	↨	'	7	G	W	g	w		ç	ù	º	À	Ã	Î	þ	¸
-8	◘	↑	(	8	H	X	h	x		ê	ÿ	¿	©	╚	Ï	Þ	°
-9	○	↓	)	9	I	Y	i	y		ë	Ö	®	╣	╔	┘	Ú	••
-A	◙	→	*	:	J	Z	j	z		è	Ü	¬	╣	╩	┌	Û	•
-B	♂	←	+	;	K	[	k	{		ï	ø	½	╗	╦	■	Ù	1
-C	♀	∟	,	<	L	\	l	\|		î	£	¼	╝	╠	▬	ý	3
-D	♪	↔	-	=	M	]	m	}		ì	Ø	¡	¢	═	¦	Ý	2
-E	♫	▲	.	>	N	^	n	~		Ä	×	«	¥	╬	Ì	¯	■
-F	☼	▼	/	?	O	_	o	⌂		Å	ƒ	»	┐	¤	▬	´	

Figure 39-4. Code Page 850

Code Page Supported Devices

It is possible to switch code pages only on hardware that allows the code page to be downloaded from OS/2 to the device. If not provided by downloading, code-page support must be switched manually at the computer. The following hardware is required to allow OS/2 to switch code pages:

- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- IBM Personal System/2 Display Adapter
- IBM Personal System/2 Display Adapter 8514/A
- IBM Proprinter.
- IBM Quietwriter III.

The switching provided by OS/2 is between one of the existing supported code pages (CP437, CP860, CP863, and CP865) and the new multilingual code page, CP850.

Note: Without the required hardware, you are restricted to the native language code page provided in the hardware ROM (CP437 in the U.S.A.).

Application Design Considerations

When deciding whether to design your application to switch code pages, consider the following:

- Performance is an important consideration when code-page switching. Frequent code-page switching can result in a reduced performance.
- Device code-page support is provided only for printer and display devices with a code page download and switch capability.

 Note: See the section in this chapter on hardware requirements.
- The application is responsible for verifying that the code page and the application agree.
- OS/2 provides printer code-page switching when a spooler is installed and DEVINFO is specified for the printer in the CONFIG.SYS file.

 OS/2 supports the following printer IOCtl commands:

 - Activate a printer code page
 - Get the active printer code page
 - Validate a printer code page.

Special Considerations and Limitations

Code Page Management has these special considerations and limitations:

- One system country code is prepared at system initialization and cannot be changed at run time; however, information for other country codes can be retrieved through the Country API for country information.
- One or two system code pages may be prepared at system initialization. The user can switch (CHCP) between them. A program through the API, described in the following section, can also switch between them. The system code pages are prepared based on CONFIG.SYS commands and system initialization and cannot be changed at run time.
- One system keyboard layout is prepared at system initialization and can be changed at run time using the KEYB utility. The changed keyboard layout replaces the current keyboard translate tables for the system code pages as defined by the CODEPAGE statement in CONFIG.SYS (if the selected layout is available in those code pages).
- The CODEPAGE statement allows preparation for up to two selected code pages, but the system may actually have more than two different code pages active in the system in these instances:
 - System Initialization defaults to a code page different from the selected code page for a printer character set, display character set, keyboard layout or the country information when the selected code page is not available.
 - KEYB defaults to a different code page for the keyboard layout when it is not available in a selected code page.
 - An application uses printer IOCtl to activate additional printer code pages.
- The CHCP command changes only the code page of the command process to which it belongs and does not affect the code page of any other existing command process, other process, or other subsystem handle. Its purpose is to allow the user to change the code page of the session.

- The user can switch between two prepared code pages with the CHCP command and the application can switch between code pages with the appropriate APIs. The primary (default) system code page is used throughout the system and by all applications if a switch never occurs following system initialization. If no system code pages have been prepared, the system runs with the default code page available for the device, country information, and keyboard layout.
- An application controls the appearance of its screen display characteristics when using OS/2 APIs to switch code pages. Characters may not appear the same for different code pages. The application can offer code-page selection to the user, if required, through its own application user interface.
- The OS/2 system spooler must be installed for printer code-page switching or be replaced by another printer monitor that receives code-page commands from the printer device driver and provides the code-page function support.
- It is possible for an application to issue printer IOCtl commands to activate different code pages throughout its printed data stream. No data stream format is defined by OS/2. Imbedded escape sequences and printer control codes are allowed and not monitored by OS/2.
- OS/2 does not automatically switch to the code page of a file name and does not keep track of the code page of a file name. This allows the following possible situations to occur:
 - The file is not accessible under a different code page because the file name is not recognized in that code page, although valid in the original code page in which it was created.
 - The wrong file becomes accessible under a different code page because the keyboard entered file name maps to a different but valid file name in that code page.
- The system does not automatically switch to the code page of a message before output. Therefore, messages encoded in a particular code page may not be fully readable in another code page.
- Make sure your application will run on machines that do not support code-page switching.
- Restrict graphics characters to those in the common character set.
- If possible, do not reserve code points for control purposes. If you must, make sure the code points you use are not alphabetic characters on any of the code pages.
- Isolate all *folding* and *monocasing* to a single routine with a well-defined interface. Whenever possible, let OS/2 perform folding.

 Note: *Folding* is the mapping of one (usually larger) character set into another (usually smaller) one. *Monocasing*, another common use of folding, is the mapping of mixed case text into either uppercase or lowercase.
- Make sure that the format of items such as dates, times, and currencies, which differ from country to country, are selectable.
- Design the routine so that it is entirely table driven. A table-driven directory can be adapted for use with another code page or in another country.
- Make the tables accessible to the user so that, if necessary, they can be overridden.
- Isolate words using a table of valid delimiters that are not code-page specific. Avoid using a table of valid alphabetics; the table then becomes code-page dependent.
- Avoid duplicating coding effort by allowing vendors to write conversion utilities for the most popular PC programs.
- Because there is no perfect mapping from CP437 to CP850, make any conversion utility table-driven and make the tables editable by the user. Or, use code points that are common to both code pages.
- Ensure that all mapping is done through tables that are accessible to the user.
- Do not dynamically switch code pages within an application with the ability to display data from more than one file at the same time. It is most important to keep screen and keyboard in agreement. If dynamic code-page switching must be used, make sure that no problems are caused by switching the printer and keyboard simultaneously.
- Do not provide dynamic data conversion in any application that allows data to be altered.

- When possible, let the user select file names from a scrollable directory rather than entering them on the keyboard. The directory should be sortable by name, date, and size.
- Design any additional text support for the graphics mode to include all the new characters in CP850.
- Design font switching so that a subset of the complete character set can be altered.

 Because 75% of the font is unchanged when code-page switching occurs, it makes sense to switch only those characters that need altering.

Code Page API Summary

The code page function calls are summarized as follows:

DosGetCp
Get the code page of the calling process and the system code page

DosSetCp
Set the code page of the calling process and the session's display and keyboard code page

DosSetProcCp
Set the code page of the calling process

GpiCreateLogFont
Create a font in a code page

GpiQueryCp
Query default GPI code page

GpiSetCp
Set default GPI code page

KbdSetCp
Set a keyboard subsystem code page

KbdGetCp
Get a keyboard subsystem code page

Printer IOCtl
Requires Print Spooler to be installed

VioCreateLogFont
Create font for use in a VIO presentation space

VioGetCp
Get a video subsystem code page

VioSetCp
Set a video subsystem code page

- Activate a printer code page
- Get the active printer code page
- Validate a printer code page.

WinCpTranslateChar (WUTRCH)
Translates a character from one code page to another

WinCpTranslateString (WUTRST)
Translates a string from one codepage to another

WinQueryCp (WMGECP)
Query code page for translating incoming messages

WinQueryCpList (WMQCPL)
Get a list of available code pages

WinSetCp (WMSECP)
Sets the code page

For function call details, see *Control Program Programming Reference*.

Chapter 40. Migrating Applications to the Presentation Manager

This chapter discusses migration of different application types to the windowing environment of the Presentation Manager.

When migrating an application to the Presentation Manager, you should ensure that the user interface portion of the application conforms with the Common User Access interface of Systems Application Architecture.

The Decision to Migrate

Users already have applications written for the DOS and OS/2 1.0 environments that they can continue to use, along with the Presentation Manager generation of applications as they are developed.

Of these existing applications, you will want to assess which ones qualify for migration to the windowed, multitasking environment provided by the Presentation Manager and OS/2.

Some application types are obvious candidates for migration. These types are:

- Alphanumeric PC-DOS Microsoft Windows applications
- Alphanumeric OS/2 1.0 applications.

Migrating PC-DOS Applications

For most PC-DOS applications, converting to the Presentation Manager windowing environment involves rewriting the application. For these applications, a migration to a text-windowed application can be considered. Text-windowed applications use a default window and a default presentation space provided by the Presentation Manager. See "Migrating OS/2 1.0 Applications to Text-Windowed Applications" on page 40-3 for a description of the requirements for a text-windowed application.

A PC-DOS application written to run under Microsoft Windows is a special case. Because a Microsoft Windows application has the same structure as a Presentation Manager application, it can migrate directly to a Presentation Manager application.

Migrating a Microsoft Windows Application

The following assessments should be made when considering migration of an application from the Microsoft Windows PC-DOS environment to the OS/2 Presentation Manager environment:

- What are the changes required for the user interface portion of the application?
- What are the changes required for the processing portion of the application?
- To what extent does the application use GDI graphics?

Changes to the User Interface

The structure of a Presentation Manager application is the same as the structure of a Microsoft Windows 2 application. Both are object-oriented, event-driven programs, consisting of an initialization phase and a message-processing loop. The initialization phase has been extended for the Presentation Manager.

Migrating the user interface portion of a Microsoft Windows application to a Presentation Manager application, involves a lot of minor changes. For example, the names for the Presentation Manager windowing calls are different from those for Microsoft Windows applications. Some of their parameters are different, both in ordering and in function.

There has been some rationalization of the Presentation Manager API calls, and of the way dialog boxes are handled. In addition, there are some naming and parameter differences in the resource file.

The methods of file and application management used by Microsoft Windows and the Presentation Manager are quite different.

Conversion from the DOS Environment

A PC-DOS application is written to run in a single-tasking environment and uses DOS interrupts. In some cases, interrupts can be easily mapped to OS/2 function calls. In others, functions are not as directly related. For a list of PC-DOS interrupts and their OS/2 counterparts, see Appendix F, "PC-DOS Interrupts and Related OS/2 Function Calls."

The Base Video Subsystem provides a broad range of function calls for displaying alphanumeric data on the screen.

Depending upon the requirements of your application, you can place alphanumeric data in the buffer using either of the following two methods:

- Directly accessing the logical video buffer with VioGetBuf, manipulating the data, and flushing it to the screen with VioShowPS. See "Directly Accessing the Logical Video Buffer" on page 30-3.
- Performing string updates to the screen with high-function Vio read and write calls (see "Performing String Updates to the Logical Video Buffer" on page 30-4). String updates should be used when only specific parts of the buffer need updating.

Migrating Microsoft Windows Graphics Applications

Migration of a Microsoft Windows application is not trivial if the application uses the GDI graphics of Microsoft Windows.

The graphics functions of the Presentation Manager are based on the graphics interfaces to be found in the IBM Graphical Data Display Manager (GDDM) and the Graphics Control Program (GCP). Under the Presentation Manager, the functions of these products have been extended to include those provided by the Graphics Device Interface (GDI) found in the Microsoft Windows product. In addition, there are some graphics functions that are not present in any of these products.

Although there are many similarities between the SAA Presentation Interface implemented by the Presentation Manager and the above products, existing applications will require rewriting. However, just as programmers with experience in writing Microsoft Windows applications will be familiar with the structure of Presentation Manager applications, so will programmers with experience in writing GDDM applications be familiar with the graphics concepts used.

Migrating OS/2 Full-Screen Applications to Presentation Manager Applications

The WINABLE utility allows you to convert many OS/2 full-screen applications so that they can run in a Presentation Manager window. WINABLE changes the format of the executable file into a text-window type of application, which is recognized by the Presentation Manager.

If the application contains API calls that cannot be converted, you will receive a warning message. However, you can include a command option telling WINABLE to continue the conversion even though the application contains unacceptable calls.

WINABLE command options also allow you to convert a file on which you have run WINABLE back to its original format or to create a backup of the original file. If the backup option is used, WINABLE creates an identical backup copy of the original executable file. If applicable, users should ensure that this operation is consistent with the terms and conditions of the license agreement under which the application was acquired.

To run WINABLE, type the following command:

```
[drive1][path1] WINABLE [drive2][path2] filename
[/u] [/b] [/c]
```

where:

drive1	is the drive location of WINABLE
path1	is the path location of WINABLE
WINABLE	starts the WINABLE utility
drive2	is the drive location of the executable file to be converted
path2	is the path location of the executable file to be converted
filename	is the name of the executable file to be converted
/u	changes an application on which you have run WINABLE back to its original format

/b creates a backup of the original executable file

/c tells WINABLE to continue converting the file even if unacceptable API calls are found.

Table 40-1 shows the API calls that cannot be converted by the WINABLE utility.

Table 40-1. API calls that are unacceptable to the WINABLE utility.

KbdDeRegister	KbdGetStatus
KbdRegister	KbdSetFgnd
KbdSetStatus	MouClose
MouDeRegister	MouDrawPtr
MouGetDevStatus	MouGetPtrPos
MouGetPtrShape	MouGetScaleFact
MouOpen	MouRegister
MouRemovePtr	MouSetDevStatus
MouSetPtrPos	MouSetScaleFact
MouSynch	VioAssociate
VioCreatePs	VioCreateLogFont
VioDeRegister	VioDeleteSetId
VioDestroyPs	VioGetDeviceCellSize
VioGetFont	VioGetMode
VioGetOrg	VioGetPhysBuf
VioGetState	VioModeUndo
VioModeWait	VioPrtSc
VioPrtScToggle	VioSetMode
DosMonClose	DosMonOpen
DosMonRead	DosMonReg
DosMonWrite	

Migrating OS/2 1.0 Applications to Text-Windowed Applications

All applications developed for OS/2 1.0 can continue to run under OS/2 1.2 as full-screen applications.

OS/2 applications that do not use I/O function calls that interact with the hardware at a physical level can be run in a default window within the OS/2 Presentation Manager screen group. Limited windowing functions, such as moving and sizing the window, are possible. This is achieved by adding the application to the Desktop Manager as a text-windowed application.

A text-windowed application uses subsets of OS/2 keyboard and mouse function calls for user input (see Chapter 31, "Keyboard and Mouse Input for Text-Windowed and Full-Screen Applications"). A text-windowed application does not explicitly create a window, nor does it use messages to communicate. When an application of this type runs, OS/2 supplies it with a default window and a default presentation space.

The default window has the following elements in common with the window described in Chapter 5, "The Standard Window":

- A system icon, from which the user can pull down a system menu.
- A title bar, containing the name of the window.
- Scroll bars, with which the user can scroll text.
- Minimize/maximize/restore icons.

A default window initially occupies 80 × 25 characters, or the entire screen, depending on which is the smaller. An application can call VioSetMode to alter the number of rows only.

The default presentation space is in CGA format.

For information on alphanumeric Vio function calls that can be used by text-windowed applications, see "Alphanumeric Support for All Application Types" on page 30-3.

Migrating OS/2 1.0 Applications to Presentation Manager Applications

The effort involved in converting an OS/2 1.0 alphanumeric Vio application to a Presentation Manager application depends on both the structure of the application and on how much use will be made of the Presentation Manager API. Conversion is a lot easier if the application has been structured so that the user interface and processing parts of the program are cleanly separated.

The full conversion of an OS/2 1.0 application to OS/2 Presentation Manager consists of two parts:

1. Restructuring the user interface part of the program into an event-driven program, which listens and reacts to incoming messages (including keyboard and mouse messages).
2. Converting the user interface from an alphanumeric Vio interface to one that fully exploits the

SAA CUA user interface possible with Presentation Manager.

You can implement only the first part of this conversion to significantly reduce the migration effort involved. A set of Vio calls specific to this type of Presentation Manager application are available (see "Alphanumeric Support for Presentation Manager Applications" on page 30-1).

Restructuring the Application

The user interface part of the program must be converted into an event-driven, Presentation Manager application. This means that:

- The application structure must be changed to a Presentation Manager application structure; that is, a main procedure and a window procedure.
- The main procedure must initialize its communication with the Presentation Manager, as in a standard PM application.
- All mouse and keyboard interaction must be handled by reacting to incoming messages to the window procedure, rather than by explicit Mou and Kbd function calls.

Migrating the Vio Calls to the Presentation Manager

The parts of the application that control output to the screen can be transferred to the section of the application that handles redrawing the screen, with minor modifications:

- A Vio Presentation Space must be created and associated with the device context for the window in which the output will be displayed.
- All Vio calls use the handle of the Vio presentation space, rather than the null handle used in OS/2 1.0 Vio calls.
- Multiple Vio presentation spaces are possible.

Vio calls that directly access the physical buffer are not allowed.

Chapter 41. Non-Program Products That You Can Develop Using OS/2

In addition to application programs, this chapter describes some of the other machine-loadable products that you can develop using OS/2:

- Function libraries
- Resource libraries
- Graphics Libraries
- Font libraries
- Installation programs.

Function Libraries

In the same way that the functions in the OS/2 API are supplied in dynamic link libraries, you can place routines in dynamic link libraries. For example, you could create a mathematical function library. You can put your code inside a window procedure and register it as a public window class. (See "Registering a Public Window Class" on page 13-9.) Any application running under a system that had loaded your function library could use a function by creating a window of your class and sending messages to it.

Window Libraries

Using the GPI, you can create your own windows, with whatever appearance and usability characteristics you want. Again, you can register such a window as a public window class.

Resource Libraries

You can use the icon editor to produce libraries of icons, pointers, and bit maps for use by applications.

Graphics Libraries

You could produce libraries of graphics pictures, stored as bit maps or metafiles.

Loadable Font Libraries

If you are a professional font designer, you can use the programming tools to produce one or more libraries of fonts. These could then be used by other products, also developed using OS/2, that can load the fonts. Business graphics packages and desktop publishing packages are two examples of such products.

You can use the font editor to produce the fonts in a resource file. The process of turning the resource file into a .DLL is outlined in "The Font Editor" on page 8-8, and described in the *Building Programs* manual. You can leave the user to install the font on the system, using the Control Panel, or supply an installation program to install the .DLL.

Installation Programs

Installation is a necessary chore with most products. Typically, a product exists on a set of diskettes, and must be loaded onto the hard disk. Often, it must also be customized in some way. Altogether, this can be a complex process, but you can make life easier for the user by writing a small installation program to do most of the hard work for PM-developed products.

An installation program for a PM product is, typically, itself a PM application that you write in addition to the main product. It can have a name like "INSTALL" so that the user can easily recognize it, and has a user interface that allows the user to perform the installation tasks. The tasks that can be performed by a PM installation program are as follows:

1. Select features and options. A typical option is the language version required.
2. Put the product files onto the hard disk. Generally, the files are put into directories which are created or specified by the user. You should give the user control over this, as the machine's hard disk may be set up in a special way.

3. Register the product on the system, for example:

 A program should be registered on the installed list. The program title will be displayed in "Desktop Manager" in the user shell, providing an easy way to start the program. You should allow the user to choose a program name or a default. You can place the data in the system using the WinAddProgram (WMADPR) call. In the data, you can include the initial size and position of the program's main window, and any environment data required by the program when it starts.

 - A font .DLL should be registered under PM_Fonts on OS2.INI, as described in "The Initialization Files" on page 14-1.
 - A non-font .DLL should be registered under SYS_DLLS on OS2.INI, as described in "Registering a Public Window Class" on page 13-9.

4. Install any product-specific data in the OS2.INI file. Rather than keeping any additional files containing option information, it is possible to use the initialization file as a place for such data. Your application can retrieve it at run time.

Appendixes

Appendix A. API Error Processing

Control Program API Processing

Successful completion of a Control Program API function is indicated by an error return code of zero.

In the event of an error, a Control Program API function call usually returns an error code that has a non-zero integer value. The non-zero value equates to a symbolic error identifier in the Control Program API include file BSEERR.H. The symbolic identifiers indicate the cause of the error. For example, a return code of 2 on the DosOpen call equates to the symbolic identifier ERROR_FILE_NOT_FOUND. The symbolic identifiers and their integer equivalents are listed in the *Control Program Programming Reference.*

OS/2 supplies the following calls for error processing:

DosErrClass
This function deals with the return codes received from the OS/2 API. The input is the error code returned from another function. The output is a classification of the error and a recommended action. Depending on the application, the recommended action could be followed, or a specific recovery action could be done.

When called by a family mode application, this function can only return a valid error classification for errors that have actually occurred. Also a classification of a given return code is not guaranteed to be identical for FAPI and protect mode applications.

DosError
Hard error processing will normally occur without direct application notification. However, if an OS/2 application wishes to process these events, the DosError call provides the services.

This function allows the application program to notify OS/2 that all permanent errors associated with the process, or for an open handle belonging to the process, are reflected as immediate failures. OS/2 presents hard errors as pop-up windows.

DosError also allows an OS/2 process to disable end user notification if a program (or untrapped numeric processor) exception occurs. If end user notification is disabled, and one of these exceptions occurs, the process will still be terminated. If DosError is not issued, the default is both hard error pop-ups and exception pop-ups enabled.

DosSetSigHandler
To receive a hard error notification from a remote program, OS/2 sends the termination signal SIG_TERM. To allow a process to receive a signal generated by specific trap and external events, issue DosSetSigHandler. For additional information on signal communication, see "Setting Flags to Respond to Time-Critical Events" on page 35-6.

DosSetVec
An application may provide a routine to process machine exceptions which occur while it is executing. This allows language libraries to furnish 287 emulation or error recovery routines.

DosSetVec allows a process to register an address to be used when an 80286 machine exception occurs. The process is analogous to setting an address in the interrupt vector table when running in the 80286 mode.

Should an exception occur and the process has registered a handler, that handler is entered as if its address had been stored in the CPU interrupt vector, except that interrupts will still be enabled. If no address has been registered for that vector then the process will be terminated.

Presentation Manager API Processing

By contrast with the Control Program functions that return zero for successful completion, some of the calls in the Presentation Manager API return a value of one for successful completion, and some of them return a value that you can use in your application. For example, the WinCreateStdWindow (WICRTS) call returns a window handle when it completes successfully.

In the event of an error, the majority of the calls in the Presentation Manager API return an error return code that has a value of zero. In addition, OS/2 puts error information in the anchor block of the thread from which WinInitialize (WMINIT) and the erroneous call were made. (In multi-threaded applications there can be multiple anchor blocks, reserved by a WinInitialize in each thread.) Note that the anchor block holds information about the *most recent* error. Several functions may have been called successfully since the error occurred.

To access the information in the anchor block a Presentation Manager application can issue the following calls:

WinGetLastError (WMGLE)
This call returns an unsigned long value.

The lower half contains the hexadecimal value of the last non-zero error code returned by a Presentation Manager call. The hexadecimal value equates to a symbolic identifier in the Presentation Manager API include file. As in the base, the symbolic identifier indicates the cause of the error. For example, the hexadecimal value 01001h equates to the symbolic identifier PMERR_INVALID_HWND, indicating that you supplied an invalid window handle as an input parameter to the call. Presentation Manager symbolic identifiers are listed in the *Control Program Programming Reference*.

Because the Presentation Manager API calls the base API inside OS/2, it is possible that, when calling a Presentation Manager function, an error can occur that is caused by a call in the base API. If this is the case, the returned hexadecimal value from WinGetLastError equates to PMERR_BASE_ERROR, that indicates a base error, and the base return value is available from WinGetErrorInfo (WMGEI).

The upper half can contain zero (for no error) or a hexadecimal value that equates to symbolic names in the OS2DEF.H and OS2DEF.INC include files, as follows:

SEVERITY_WARNING
The function detected a problem but took some remedial action and was able to complete successfully.

SEVERITY_ERROR
The function detected a problem for which no sensible remedial action is possible. The function is not executed, and the system remains at the same state as before the function was requested.

SEVERITY_SEVERE
The function detected a problem from which the system cannot reestablish its state. The function has partially executed, and the application must perform some corrective action to restore the system to a known state.

SEVERITY_UNRECOVERABLE
The function detected a problem from which the system cannot recover to the state that existed before the function was called. It is also impossible for the application to restore the system to a known state.

WinGetLastError sets the error code in the anchor block to zero, and should be called before the call being tested for errors (to clear the most recent error) as well as after.

WinGetErrorInfo (WMGEI)
This call returns an error information block that has the structure ERRINFO. The structure contains the following information:

- A field, idError, that contains the same information returned by WinGetLastError (WMGLE).
- A field, offBinaryData, that, when the Presentation Manager error is PMERR_BASE_ERROR, contains an offset to the base OS/2 error code.
- A field, offaoffszMsg, that contains an offset to a message string. The string contains the severity level preceded by the five-digit decimal error code, and one of the following component prefixes:

 DDI Device drivers

 GRE Engine

 GPI Graphics Programming Interface

 SPL Spooler

 WIN User

 SHL Shell

 UNK Undefined.

The error code in the message string is the decimal representation of the least significant three digits of the complete four-digit hexadecimal error code. The offsets returned in the ERRINFO structure can be used with the selector portion of the ERRINFO pointer. An application can use WinGetErrorInfo (WMGEI) to obtain the ERRINFO data even after WinGetLastError (WMGLE) has cleared the error. WinGetErrorInfo should always be followed by WinFreeErrorInfo (WMFERI). This releases the memory allocated for the error information block.

Appendix B. GPI Calls

A number of the GPI error conditions indicate that a call has been used in the wrong context. This appendix lists every call and shows, for each one, whether it can be used:

- In a micro presentation space
- While there is an open segment bracket
- While there is an open area bracket
- While there is an open element bracket
- While there is an open path bracket.

A check mark (√) means that a call can be used, and a cross (X) means that it cannot. There are some additional qualifiers in the form of superscript numbers. These generally indicate some further restriction on the context in which a call can be issued, and are explained at the end of this appendix on page B-8.

Table B-1 (Page 1 of 7). Which GPI Calls Where?

GPI Call	Micro-PS	Segment Bracket	Area Bracket	Element Bracket	Path Bracket
GpiAssociate	X	√	X[7]	√[6]	X[7]
GpiBeginArea	√	√	X	√	X
GpiBeginElement	X	√	√	X	√
GpiBeginPath	√	√	X	√	X
GpiBitBlt	√	√	X	√	X
GpiBox	√	√	√	√	√
GpiCallSegmentMatrix	X	√	√	√	√
GpiCharString	√	√	X	√	√
GpiCharStringAt	√	√	X	√	√
GpiCharStringPos	√	√	X	√	√
GpiCharStringPosAt	√	√	X	√	√
GpiCloseFigure	√	√	X	√	√
GpiCloseSegment	X	√	X[7]	√	X[7]
GpiCombineRegion	√	√	X	√	X
GpiComment	√	√	√	√	√
GpiConvert	√	√	√	√	√
GpiCopyMetaFile	√	√	√	√	√
GpiCorrelateChain	X	√	X	X	X
GpiCorrelateFrom	X	√	X	X	X
GpiCorrelateSegment	X	√	X	X	X
GpiCreateBitmap	√	√	√	√	√
GpiCreateLogColorTable	√	√	X	√	X
GpiCreateLogFont	√	√	X	X	√
GpiCreatePS	—	—	—	—	—
GpiCreateRegion	√	√	X	√	X

Table B-1 (Page 2 of 7). Which GPI Calls Where?

GPI Call	Micro-PS	Segment Bracket	Area Bracket	Element Bracket	Path Bracket
GpiDeleteBitmap	√	√	√	√	√
GpiDeleteElement	X	√[3]	√[3]	X	√[3]
GpiDeleteElementRange	X	√[3]	√[3]	X	√[3]
GpiDeleteElementsBetweenLabels	X	√[3]	√[3]	X	√[3]
GpiDeleteMetaFile	√	√	√	√	√
GpiDeleteSegment	X	√	√	√	√
GpiDeleteSegments	X	√	√	√	√
GpiDeleteSetId	√	√	X	√	√
GpiDestroyPS	√[1]	√	√	√	√
GpiDestroyRegion	√	√	X	√	X
GpiDrawChain	X	√	X	X	X
GpiDrawDynamics	X	√	X	X	X
GpiDrawFrom	X	√	X	X	X
GpiDrawSegment	X	√	X	X	X
GpiElement	X	√	√	X	√
GpiEndArea	√	√	√	√	X
GpiEndElement	X	√	√	√	√
GpiEndPath	√	√	X	√	√
GpiEqualRegion	√	√	X	√	X
GpiErase	√	√	X	√	X
GpiErrorSegmentData	X	√	√	√	√
GpiExcludeClipRectangle	√	√	X	√	X
GpiFillPath	√	√	X	√	X
GpiFullArc	√	√	√	√	√
GpiGetData	X	√[5]	√	√	√
GpiImage	√	√	X	√	X
GpiIntersectClipRectangle	√	√	X	√	X
GpiLabel	X	√	√	X	√
GpiLine	√	√	√	√	√
GpiLoadBitmap	√	√	√	√	√
GpiLoadFonts	√	√	√	√	√
GpiLoadMetaFile	√	√	√	√	√
GpiMarker	√	√	X	√	√
GpiModifyPath	√	√	X	√	X

Table B-1 (Page 3 of 7). Which GPI Calls Where?					
GPI Call	**Micro-PS**	**Segment Bracket**	**Area Bracket**	**Element Bracket**	**Path Bracket**
GpiMove	√	√	√	√	√
GpiOffsetClipRegion	√	√	X	√	X
GpiOffsetElementPointer	X	√[3]	√[3]	X	√[3]
GpiOffsetRegion	√	√	X	√	X
GpiOpenSegment	X	X	X[7]	√[6]	X[7]
GpiOutlinePath	√	√	X	√	X
GpiPaintRegion	√	√	X	√	X
GpiPartialArc	√	√	√	√	√
GpiPlayMetaFile	√	X	X	X	X
GpiPointArc	√	√	√	√	√
GpiPolyFillet	√	√	√	√	√
GpiPolyFilletSharp	√	√	√	√	√
GpiPolyLine	√	√	√	√	√
GpiPolyMarker	√	√	X	√	√
GpiPolySpline	√	√	√	√	√
GpiPop	X	√	√	√	√
GpiPtInRegion	√	√	X	√	X
GpiPtVisible	√	√	X	√	X
GpiPutData	X	√	√	√	√
GpiQueryArcParams	√	√[2]	√[2]	√[2]	√[2]
GpiQueryAttrMode	X	√	√	√	√
GpiQueryAttrs	√	√[2]	√[2]	√[2]	√[2]
GpiQueryBackColor	√	√[2]	√[2]	√[2]	√[2]
GpiQueryBackMix	√	√[2]	√[2]	√[2]	√[2]
GpiQueryBitmapBits	√	√[4]	X	√	X
GpiQueryBitmapDimension	√	√	√	√	√
GpiQueryBitmapHandle	√	√	√	√	√
GpiQueryBitmapParameters	√	√	√	√	√
GpiQueryBoundaryData	√	√	√	√	√
GpiQueryCharAngle	√	√[2]	√[2]	√[2]	√[2]
GpiQueryCharBox	√	√[2]	√[2]	√[2]	√[2]
GpiQueryCharDirection	√	√	X	√	√
GpiQueryCharMode	√	√[2]	√[2]	√[2]	√[2]
GpiQueryCharSet	√	√[2]	√[2]	√[2]	√[2]

Table B-1 (Page 4 of 7). Which GPI Calls Where?					
GPI Call	**Micro-PS**	**Segment Bracket**	**Area Bracket**	**Element Bracket**	**Path Bracket**
GpiQueryCharShear	√	√[2]	√[2]	√[2]	√[2]
GpiQueryCharStringPos	√	√[2]	√[2]	√[2]	√[2]
GpiQueryCharStringPosAt	√	√[2]	√[2]	√[2]	√[2]
GpiQueryClipBox	√	√	√	√	√
GpiQueryClipRegion	√	√	√	√	√
GpiQueryColor	√	√[2]	√[2]	√[2]	√[2]
GpiQueryColorData	√	√	√	√	√
GpiQueryColorIndex	√	√	√	√	√
GpiQueryCp	√	√	√	√	√
GpiQueryCurrentPosition	√	√[2]	√[2]	√[2]	√[2]
GpiQueryDefaultViewMatrix	√	√	√	√	√
GpiQueryDefCharBox	√	√	√	√	√
GpiQueryDevice	√	√	√	√	√
GpiQueryDeviceBitmapFormats	√	√	√	√	√
GpiQueryDrawControl	√	√	√	√	√
GpiQueryDrawingMode	X	√	√	√	√
GpiQueryEditMode	X	√	√	√	√
GpiQueryElement	X	√[3]	√[3]	√[3]	√[3]
GpiQueryElementPointer	X	√[3]	√[3]	√[3]	√[3]
GpiQueryElementType	X	√[3]	√[3]	√[3]	√[3]
GpiQueryFontFileDescriptions	√	√	√	√	√
GpiQueryFontMetrics	√	√	√	√	√
GpiQueryFonts	√	√	√	√	√
GpiQueryGraphicsField	√	√	√	√	√
GpiQueryInitialSegmentAttrs	X	√	√	√	√
GpiQueryKerningPairs	√	√	√	√	√
GpiQueryLineEnd	√	√[2]	√[2]	√[2]	√[2]
GpiQueryLineJoin	√	√[2]	√[2]	√[2]	√[2]
GpiQueryLineType	√	√[2]	√[2]	√[2]	√[2]
GpiQueryLineWidth	√	√[2]	√[2]	√[2]	√[2]
GpiQueryLineWidthGeom	√	√[2]	√[2]	√[2]	√[2]
GpiQueryLogicalColorTable	√	√	√	√	√
GpiQueryMarker	√	√[2]	√[2]	√[2]	√[2]
GpiQueryMarkerBox	√	√[2]	√[2]	√[2]	√[2]

Table B-1 (Page 5 of 7). Which GPI Calls Where?					
GPI Call	**Micro-PS**	**Segment Bracket**	**Area Bracket**	**Element Bracket**	**Path Bracket**
GpiQueryMarkerSet	√	√[2]	√[2]	√[2]	√[2]
GpiQueryMetaFileBits	√	√	√	√	√
GpiQueryMetaFileLength	√	√	√	√	√
GpiQueryMix	√	√[2]	√[2]	√[2]	√[2]
GpiQueryModelTransformMatrix	√	√[2]	√[2]	√[2]	√[2]
GpiQueryNearestColor	√	√	√	√	√
GpiQueryNumberSetIds	√	√	√	√	√
GpiQueryPageViewPort	√	√	√	√	√
GpiQueryPattern	√	√[2]	√[2]	√[2]	√[2]
GpiQueryPatternRefPoint	√	√[2]	√[2]	√[2]	√[2]
GpiQueryPatternSet	√	√[2]	√[2]	√[2]	√[2]
GpiQueryPel	√	√	√	√	√
GpiQueryPickAperturePosition	√	√	√	√	√
GpiQueryPickApertureSize	√	√	√	√	√
GpiQueryPS	√	√	√	√	√
GpiQueryRealColors	√	√	√	√	√
GpiQueryRegionBox	√	√	√	√	√
GpiQueryRegionRects	√	√	√	√	√
GpiQueryRGBColor	√	√	√	√	√
GpiQuerySegmentAttrs	X	√	√	√	√
GpiQuerySegmentNames	X	√	√	√	√
GpiQuerySegmentPriority	X	√	√	√	√
GpiQuerySegmentTransformMatrix	X	√	√	√	√
GpiQuerySetIds	√	√	√	√	√
GpiQueryStopDraw	X	√	√	√	√
GpiQueryTag	X	√[2]	√[2]	√[2]	√[2]
GpiQueryTextBox	√	√[2]	√[2]	√[2]	√[2]
GpiQueryViewingLimits	√	√[2]	√[2]	√[2]	√[2]
GpiQueryViewingTransformMatrix	X	√	√	√	√
GpiQueryWidthTable	√	√	√	√	√
GpiRealizeColorTable	√	√	X	√	X
GpiRectInRegion	√	√	X	√	X
GpiRectVisible	√	√	X	√	X
GpiRemoveDynamics	X	√	X	X	X

Table B-1 (Page 6 of 7). Which GPI Calls Where?					
GPI Call	**Micro-PS**	**Segment Bracket**	**Area Bracket**	**Element Bracket**	**Path Bracket**
GpiResetBoundaryData	√	√	√	√	√
GpiResetPS	√	√	√[6]	√[6]	√[6]
GpiRestorePS	√	√[4]	X	√[4]	X
GpiRotate	√	√	√	√	√
GpiSaveMetaFile	√	√	√	√	√
GpiSavePS	√	√[4]	X	√[4]	X
GpiScale	√	√	√	√	√
GpiSetArcParams	√	√	√	√	√
GpiSetAttrMode	X	√	√	√	√
GpiSetAttrs	√	√	X	√	√
GpiSetBackColor	√	√	X	√	X
GpiSetBackMix	√	√	X	√	X
GpiSetBitmap	√	√	X	√	X
GpiSetBitmapBits	√	√	X	√	X
GpiSetBitmapDimension	√	√	√	√	√
GpiSetBitmapId	√	√	√	√	√
GpiSetCharAngle	√	√	X	√	√
GpiSetCharBox	√	√	X	√	√
GpiSetCharDirection	√	√	X	√	√
GpiSetCharMode	√	√	X	√	√
GpiSetCharSet	√	√	X	√	√
GpiSetCharShear	√	√	X	√	√
GpiSetClipPath	√	√	X	√	X
GpiSetClipRegion	√	√	X	√	X
GpiSetColor	√	√	X	√	√
GpiSetCp	√	√	X	√	√
GpiSetCurrentPosition	√	√	√	√	√
GpiSetDefaultViewMatrix	√	√	X	√	X
GpiSetDrawControl	√	X	X	X	X
GpiSetDrawingMode	X	X	X	X	X
GpiSetEditMode	X	√	√	X	√
GpiSetElementPointer	X	√[2]	√[2]	X	√[2]
GpiSetElementPointerAtLabel	X	√[2]	√[2]	X	√[2]
GpiSetGraphicsField	√	√	X	√	X

Table B-1 (Page 7 of 7). Which GPI Calls Where?

GPI Call	Micro-PS	Segment Bracket	Area Bracket	Element Bracket	Path Bracket
GpiSetInitialSegmentAttrs	X	√	√	√	√
GpiSetLineEnd	√	√	X	√	√
GpiSetLineJoin	√	√	X	√	√
GpiSetLineType	√	√	X	√	√
GpiSetLineWidth	√	√	X	√	√
GpiSetLineWidthGeom	√	√	X	√	X
GpiSetMarker	√	√	X	√	√
GpiSetMarkerBox	√	√	X	√	√
GpiSetMarkerSet	√	√	X	√	√
GpiSetMetaFileBits	√	√	√	√	√
GpiSetMix	√	√	X	√	√
GpiSetModelTransformMatrix	√	√	√	√	√
GpiSetPageViewport	√	√	X	√	X
GpiSetPattern	√	√	X	√	X
GpiSetPatternRefPoint	√	√	X	√	X
GpiSetPatternSet	√	√	X	√	X
GpiSetPel	√	√	X	√	X
GpiSetPickAperturePosition	√	√	X	√	X
GpiSetPickApertureSize	√	√	X	√	X
GpiSetPS	√	√	√	√	√
GpiSetRegion	√	√	X	√	X
GpiSetSegmentAttrs	X	√	√	√	√
GpiSetSegmentPriority	X	X[5]	√	X	√
GpiSetSegmentTransformMatrix	X	√	√	√	√
GpiSetStopDraw	X	√	√	√	√
GpiSetTag	X	√	X	√	√
GpiSetViewingLimits	√	√	X	√	X
GpiSetViewingTransformMatrix	X	X	X	√	X
GpiStrokePath	√	√	X	√	X
GpiTranslate	√	√	√	√	√
GpiUnloadFonts	√	√	√	√	√
GpiUnrealizeColorTable	√	√	X	√	X
GpiWCBitBlt	√	√	√	√	√

Notes:

1. Not valid to a cached micro presentation space.
2. Valid only when the actual drawing mode is 'draw' or 'draw-and-retain'. The actual drawing mode is determined as shown in the following table:

Table B-2. The Current Drawing Mode

GpiSetDrawingMode parameter	**Context**		
	Chained Segment	**Unchained Segment**	**Outside Segment**
DM_DRAWANDRETAIN	draw-and-retain	retain	draw
DM_RETAIN	retain	retain	draw
DM_DRAW	draw	retain	draw

 For example, if the current drawing mode parameter is DM_RETAIN, and primitives are being drawn outside a segment, then the actual drawing mode is 'draw'.
3. Valid only when the actual drawing mode (see note 2) is 'retain'.
4. Valid only when the actual drawing mode (see note 2) is 'draw'.
5. Not valid if the specified segment is the current open segment.
6. Bracket (path, element, or area) is ended without error.
7. Severity is Warning.

Appendix C. Graphics Attributes

Every graphics presentation space has a set of *graphics attributes*. A normal presentation space has a larger set of graphics attributes than a micro or cached micro presentation space. (Segment-related attributes, for example, do not apply in micro presentation spaces.) These attributes all have default values, which means that they always have an effect on graphics, even if you have not explicitly specified their values. The attributes can be broken into two general groups. The first group comprises those attributes that form a part of the picture and that can vary as the picture is drawn. These are:

- All primitive attributes
- The segment attributes
- The primitive tag
- The current position
- The viewing window
- The clipping path
- The model and segment transformations
- The viewing transformation.

With the exception of the segment attributes and the viewing transformation, these attributes are reset to their default values at the start of a root segment, unless the fast-chaining attribute is set. If the fast-chaining attribute is set, their values cannot be guaranteed. You should, therefore, explicitly set any attribute values required in the segment. Similarly, the values of these attributes cannot be guaranteed following the issue of:

Any GpiDraw call
Any GpiCorrelate call
GpiCallSegmentMatrix
GpiCloseSegment.

Therefore, if any of these calls is followed by primitives outside a segment, you should explicitly set required attribute values. When GpiCloseSegment is followed by the GpiOpenSegment call (and also between any two segments in the chain), the fast-chaining attribute determines what happens to the current values of these attributes. The viewing transformation and the segment attributes are unaffected by fast-chaining.

These attributes take effect when graphics are sent to an output device, *not* when graphics are defined. For this reason, the GpiQuery calls that retrieve the current values of these attributes are invalid in retain mode. The majority of the group-one attribute-setting calls cause graphics orders to be added to the current segment.

The second group of attributes defines the environment in which the picture is drawn. These attributes do not normally vary as the picture is drawn, but have an overall effect on the result of any drawing or correlation operation. This group includes:

- The default viewing transformation
- The page viewport
- The graphics field
- The clipping region
- The pick-aperture size
- The drawing controls.

The GpiQuery calls that retrieve the current values of these attributes are valid in all drawing modes. None of the group-two attribute-setting calls causes graphics orders to be added to the current segment.

The following table lists the graphics attributes, identifies the GPI call or calls that you use to change current settings, and lists the default value of each one. Note that these default values are the initial settings. The table also indicates whether the attribute is a group-one or a group-two attribute.

Graphics Attribute	GPI Call	Default Value	Group One	Group Two
Arc parameters	GpiSetArcParams	(1,1,0,0)	√	
Foreground color	GpiSetColor[1]	CLR_DEFAULT (device-dependent)	√	
Foreground mix	GpiSetMix[1]	FM_DEFAULT (overpaint)	√	
Background color	GpiSetBackColor[1]	CLR_DEFAULT (device-dependent)	√	
Background mix	GpiSetBackMix[1]	BM_DEFAULT (leave-alone)	√	
Character angle	GpiSetCharAngle[1]	(1,0)	√	
Character box	GpiSetCharBox[1]	Device-dependent	√	
Character Direction	GpiSetCharDirection[1]	CHDIRN_DEFAULT (leftright)	√	
Character mode	GpiSetCharMode[1]	CM_DEFAULT (mode 1)	√	
Character set	GpiSetCharSet[1]	LCID_DEFAULT (system font)	√	
Character shear	GpiSetCharShear[1]	(0,1)	√	
Line end	GpiSetLineEnd[1]	LINEEND_DEFAULT (flat)	√	
Line join	GpiSetLineJoin[1]	LINEJOIN_DEFAULT (beveled)	√	
Line type	GpiSetLineType[1]	LINETYPE_DEFAULT (solid)	√	
Line width	GpiSetLineWidth[1]	LINEWIDTH_DEFAULT (1.0)	√	
Geometric line width	GpiSetLineWidthGeom[1]	1	√	
Marker	GpiSetMarker[1]	MARKSYM_DEFAULT (cross)	√	
Marker box	GpiSetMarkerBox[1]	Device-dependent	√	
Marker set	GpiSetMarkerSet[1]	LCID_DEFAULT (system marker set)	√	
Pattern	GpiSetPattern[1]	PATSYM_DEFAULT (solid)	√	
Pattern reference point	GpiSetPatternRefPoint[1]	(0,0)	√	
Pattern set	GpiSetPatternSet[1]	LCID_DEFAULT (system pattern set)	√	
Model transformation	GpiSetModelTransformMatrix	Identity	√	
Instance transformation	GpiCallSegmentMatrix	Identity	√	
Segment transformation	GpiSetSegmentTransformMatrix	Identity	√	
Viewing transformation	GpiSetViewingTransformMatrix	Identity	√	

Graphics Attribute	GPI Call	Default Value	Group One	Group Two
Default viewing transformation	GpiSetDefaultViewMatrix	Identity		√
Page viewport	GpiSetPageViewport	Device-dependent		√
Clipping path	GpiSetClipPath	No clipping	√	
Viewing window	GpiSetViewingLimits	No clipping	√	
Graphics field	GpiSetGraphicsField	No clipping		√
Clipping region	GpiSetClipRegion	No clipping		√
Tag	GpiSetTag	0	√	
Segment attributes	GpiSetInitialSegmentAttrs GpiSetSegmentAttrs	ATTR_DETECTABLE - OFF ATTR_VISIBLE - ON ATTR_CHAINED - ON ATTR_DYNAMIC - OFF ATTR_FASTCHAIN - ON ATTR_PROP_DETECTABLE - ON ATTR_PROP_VISIBLE - ON	√	
Pick-aperture size	GpiSetPickApertureSize	Device-dependent		√
Current position	GpiSetCurrentPosition GpiMove[2]	(0,0)	√	
Drawing controls	GpiSetDrawControl	DCTL_ERASE - OFF DCTL_DISPLAY - ON DCTL_BOUNDARY - OFF DCTL_DYNAMIC - OFF DCTL_CORRELATE - OFF		√

Notes:

[1] Can also be set using the GpiSetAttrs call.
[2] Is also updated indirectly by most primitive-drawing calls.

Each of the following calls causes all group-one and group-two graphics attributes to be set to their default values:

- GpiCreatePS
- GpiSetPS
- GpiResetPS, if GRES_ALL or GRES_SEGMENTS is specified
- WinGetPS
- WinGetScreenPS
- WinBeginPaint, with a NULL presentation space handle.

In specific circumstances, some of the GPI calls modify the group-one attributes and thus make their values unpredictable. Therefore, when you issue any of these calls, you should respecify attribute values that have a particular importance to your application. For example, if the current foreground color is CLR_RED before you issue the GpiDrawChain call, you cannot always rely on the current color still being CLR_RED when GpiDrawChain completes. If you want to continue working in red, respecify the color when GpiDrawChain completes. In general, the calls that affect group-one-attribute values are those related to the drawing and correlation of retained graphics, and to the creation, closing, and deletion of graphics segments. The circumstances in which these calls affect group-one graphics attributes are described below:

- The following call sets the group-one attributes[1] to their default values, and in addition sets the current clipping region and page viewport to their default values:
 - GpiAssociate.
- Each of the following calls makes the group-one-attribute values[1] unpredictable:
 - GpiCloseSegment
 - GpiCorrelateChain
 - GpiCorrelateFrom
 - GpiCorrelateSegment
 - GpiErase
 - GpiCallSegmentMatrix
 - GpiDrawDynamics[2]
 - GpiRemoveDynamics[2].
- Each of the following calls makes group-one-attribute values[1] unpredictable if a segment to be deleted is open when the call is issued:
 - GpiDeleteSegment
 - GpiDeleteSegments.
- Each of the following calls makes group-one-attribute values[1] unpredictable if there is no open segment when the call is issued:
 - GpiDrawChain
 - GpiDrawFrom
 - GpiDrawSegment.

 If there is an open segment when any of these calls is issued, and that segment was the last one drawn, then group-one-attribute values are unaffected. If, however, dynamic segments were caused to be redrawn by the same call, group-one-attribute values are made unpredictable. This occurs because dynamic segments are always drawn after nondynamic segments.

[1] With the exception of the viewing transformation and the segment attributes. The default values of these attributes apply to all primitive-drawing that occurs outside a segment bracket.

[2] Except that the foreground-mix value is always set to FM_XOR, and the background-mix value is always set to BM_LEAVEALONE.

Appendix D. Graphics Orders

In the retain and draw-and-retain drawing modes, specific GPI calls (primitive-drawing and attribute-setting calls, plus some others) cause graphics orders to be stored in the current segment. A graphics order is a sequence of one or more bytes of data that describe a graphics function. There is typically a one-to-one correspondence between a GPI call and a graphics order. You do not need to understand the various formats and contents of the graphics orders, unless:

- You are using the GpiGetData or GpiPutData calls for bulk transfer of data that you want to edit. You do not need to understand the graphics orders if you are simply copying data from one segment to another.
- You are using the GpiElement call to add data to a segment, or the GpiQueryElement call to retrieve data from a segment.
- You are examining the contents of a metafile.

Both the graphics orders and the metafile structure are described in the *Programming Reference: Volume 2*. This appendix describes the header file PMORD.H, which has been provided to allow you to manipulate the graphics orders more easily.

The Graphics-Orders Header File (PMORD.H)

A set of helper constants, macros, and structures has been provided to help you decode and encode graphics orders. These items are defined in the header file PMORD.H.

There are four types of graphics order. The first byte of each order, regardless of the graphics-order type, is the order code itself, which either partially or completely describes what follows. Depending on the order type, the graphics order can contain further information.

The four types of graphics order are:

1-Byte Order
: The 1-byte order comprises a single byte:

```
BYTE 1 : order code.
```

2-Byte Order
: The 2-byte order consists of two bytes:

```
BYTE 1 : order code
BYTE 2 : associated value.
```

Long Order
: The long order can comprise up to 257 bytes of information:

```
BYTE 1     : order code
BYTE 2     : length of order (0 to 255)
BYTE 3-257 : associated value bytes
             depending on the order code.
```

There is a special long order (Escape) where:

```
BYTE 3     : escape type
BYTE 4     : escape identifier
BYTE 5-257 : associated value bytes
             depending on the escape
             identifier.
```

Very Long Order
: The very long order can comprise up to 65537 bytes of information:

```
BYTE 1       : order code
BYTE 2       : order qualifier
BYTE 3       : length of order
               (most significant byte)
BYTE 4       : length of order
               (least significant byte -
                length of order is 0 to 65535)
BYTE 5-65537 : associated value bytes
               depending on the order
               qualifier.
```

There is a special very long order (Escape) where :

```
BYTE 5       : escape type
BYTE 6       : escape identifier
BYTE 7-65537 : associated value bytes
               depending on the escape
               identifier.
```

Decoding Graphics Orders

The recommended way of decoding a buffer of graphics orders (in C language) is to use a pointer to address the first byte of the buffer, and then retrieve the graphics order it contains. To discover which of the four types of order you have, use the following macros:

- BYTE_ORDER (1-byte order)
- SHORT_ORDER (2-byte order)
- LONG_ORDER (long order)
- VLONG_ORDER (very long order).

These macros are defined in the header file PMORD.H. Each macro processes a single byte of data and returns a Boolean value (zero or nonzero). A zero value means that the order is not of that type. When you know the graphics-order type, you can establish the length of the order, and add the length to the pointer. You can then retrieve the next order in the buffer, and repeat the process until all data has been retrieved.

You can decode the graphics-order data itself by providing a routine for each of the order types, or a routine for each individual order:

- For a 1-byte graphic order, the decoding routine should simply return a length of 1.
- For a 2-byte graphic order, the decoding routine can use the overlay structure ORDER to decode the data. The routine should return a length of 2.
- For a long order, the decoding routine can use the overlay structure LORDER to decode the data. The length of the data is a variable value.
- For a very long order, the decoding routine can use the overlay structure VORDER to decode the data. The length of the data is a variable value.

The overlay structures ORDER, LORDER, and VORDER are defined in the header file PMORD.H.

You can build graphics orders using the same structures and order types that are used for decoding graphics orders.

Naming Conventions

The names of the graphics-order codes are in the form OCODE_Gxxx. The Gxxx abbreviation is the name of the individual order, and can be used for types, structures, and constants directly related to that order. In the header file, there is a comment on the same line as each of the orders that describes the order. For example, the Begin Area order (GBAR) is described in the header file as follows:

```
#define OCODE_GBAR     0x68  /* Begin area    */
#define GBAR_BOUNDARY  0xC0
```

Notes:

1. In some structures, an S or an L is added to the name to differentiate between the short-coordinate form (16-bit) and the long-coordinate form (32-bit). For example, the Set Arc Parameters order (GSAP) is as follows:

   ```
   #define OCODE_GSAP     0x22
   #define OCODE_GPSAP    0x62

     typedef struct _ORDERS_GSAP {
       SHORT  p;
       SHORT  q;
       SHORT  r;
       SHORT  s;
     } ORDERS_GSAP;

     typedef struct _ORDERL_GSAP {
       LONG   p;
       LONG   q;
       LONG   r;
       LONG   s;
     } ORDERL_GSAP;
   ```

 In this example, the structures ORDERS_GSAP and ORDERL_GSAP are shared by GSAP (set arc parameters) and GPSAP (push and set arc parameters). As a rule, there is structure sharing between the set and push-and-set forms of graphics orders.

2. There is structure-sharing between the current-position and the given-position forms of some orders. For example, the orders GCARC (arc at current position) and GARC (arc at given position) share a structure.

Appendix E. Dos, Kbd, Mou, and Vio Calls by Application Type

The following table shows whether each Dos, keyboard (Kbd), mouse (Mou) and video (Vio) call can be used in the following:

- PM applications
- Vio-windowed applications
- Full-screen applications.

A √ symbol means that a call can be used. A X symbol means that a call cannot be used.

The final column shows whether each call is present in the family API, and can therefore be used to write DOS-mode applications. An (R) symbol means that the call works in a restricted way.

Although many of the Kbd and Mou calls are permitted in a PM application, you are recommended to use the PM message-based calls for keyboard and mouse input.

Win and Gpi calls are not listed, as they can be used only in PM applications.

For full details of any call, and any restrictions on its use, see the *Control Program Programming Reference*.

Table E-1 (Page 1 of 5). Which Dos Calls Where?

Dos Call	PM	Vio-Windowed	Full-Screen	Family API
Multitasking				
DosCreateThread	√	√	√	X
DosCWait	√	√	√	X
DosEnterCritSec	√	√	√	X
DosExecPgm	√	√	√	√(R)
DosExit	√	√	√	√(R)
DosExitCritSec	√	√	√	X
DosExitList	√	√	√	X
DosGetInfoSeg	√	√	√	X
DosGetPrty	√	√	√	X
DosKillProcess	√	√	√	X
DosPtrace	√	√	√	X
DosSetPrty	√	√	√	X
Asynchronous Notification				
DosHoldSignal	√	√	√	√(R)
DosSetSigHandler	√	√	√	√(R)
Interprocess Communication				
DosCloseQueue	√	√	√	X

Table E-1 (Page 2 of 5). Which Dos Calls Where?				
Dos Call	**PM**	**Vio-Windowed**	**Full-Screen**	**Family API**
DosCloseSem	√	√	√	X
DosCreateQueue	√	√	√	X
DosCreateSem	√	√	√	X
DosFlagProcess	√	√	√	X
DosMakePipe	√	√	√	X
DosMuxSemWait	√	√	√	X
DosOpenQueue	√	√	√	X
DosOpenSem	√	√	√	X
DosPeekQueue	√	√	√	X
DosPurgeQueue	√	√	√	X
DosQueryQueue	√	√	√	X
DosReadQueue	√	√	√	X
DosResumeThread	√	√	√	X
DosSemClear	√	√	√	X
DosSemRequest	√	√	√	X
DosSemSet	√	√	√	X
DosSemSetWait	√	√	√	X
DosSemWait	√	√	√	X
DosSuspendThread	√	√	√	X
DosWriteQueue	√	√	√	X
Timer				
DosGetDateTime	√	√	√	√
DosSetDateTime	√	√	√	√
DosSleep	√	√	√	√(R)
DosTimerAsync	√	√	√	X
DosTimerStart	√	√	√	X
DosTimerStop	√	√	√	X
Memory Management				
DosAllocHuge	√	√	√	√(R)
DosAllocSeg	√	√	√	√(R)
DosAllocShrSeg	√	√	√	X
DosCreateCSAlias	√	√	√	√(R)
DosFreeSeg	√	√	√	√
DosGetHugeShift	√	√	√	√

Table E-1 (Page 3 of 5). Which Dos Calls Where?				
Dos Call	**PM**	**Vio-Windowed**	**Full-Screen**	**Family API**
DosGetSeg	√	√	√	X
DosGetShrSeg	√	√	√	X
DosGiveSeg	√	√	√	X
DosLockSeg	√	√	√	X
DosMemAvail	√	√	√	X
DosReallocHuge	√	√	√	√(R)
DosReallocSeg	√	√	√	√(R)
DosSubAlloc	√	√	√	√
DosSubFree	√	√	√	√
DosSubSet	√	√	√	√
DosUnlockSeg	√	√	√	X
Dynamic Linking				
BadDynLink	√	√	√	√
DosFreeModule	√	√	√	X
DosGetMachineMode	√	√	√	√
DosGetModHandle	√	√	√	X
DosGetModName	√	√	√	X
DosGetProcAddr	√	√	√	X
DosLoadModule	√	√	√	X
Device Monitors				
DosMonClose	X	X	√	X
DosMonOpen	X	X	√	X
DosMonRead	X	X	√	X
DosMonReg	X	X	√	X
DosMonWrite	X	X	√	X
Session Management				
DosSelectSession	√	√	√	X
DosSetSession	√	√	√	X
DosStartSession	√	√	√	X
DosStopSession	√	√	√	X
Device I/O				
DosBeep	√	√	√	√
DosCLIAccess	√	√	√	X
DosDevConfig	√	√	√	√

Table E-1 (Page 4 of 5). Which Dos Calls Where?				
Dos Call	**PM**	**Vio-Windowed**	**Full-Screen**	**Family API**
DosDevIOCtl	√	√	√	√(R)
DosGetPID	√	√	√	√
DosPFSActivate	√	√	√	X
DosPFSCloseUser	√	√	√	X
DosPFSInit	√	√	√	X
DosPFSQueryAct	√	√	√	X
DosPFSVerifyFont	√	√	√	X
DosPhysicalDisk	√	√	√	X
DosPortAccess	√	√	√	X
DosSendSignal	√	√	√	X
File I/O				
DosBufReset	√	√	√	√
DosChDir	√	√	√	√
DosChgFilePtr	√	√	√	√
DosClose	√	√	√	√
DosDelete	√	√	√	√
DosDupHandle	√	√	√	√
DosFileLocks	√	√	√	√(R)
DosFindClose	√	√	√	√(R)
DosFindFirst	√	√	√	√(R)
DosFindNext	√	√	√	√(R)
DosMkDir	√	√	√	√
DosMove	√	√	√	√
DosNewSize	√	√	√	√
DosOpen	√	√	√	√(R)
DosQCurDir	√	√	√	√
DosQCurDisk	√	√	√	√
DosQFHandState	√	√	√	√(R)
DosQFileInfo	√	√	√	√
DosQFileMode	√	√	√	√
DosQFsInfo	√	√	√	√
DosQHandType	√	√	√	√
DosQVerify	√	√	√	√
DosRead	√	√	√	√

Table E-1 (Page 5 of 5). Which Dos Calls Where?				
Dos Call	**PM**	**Vio-Windowed**	**Full-Screen**	**Family API**
DosReadAsync	√	√	√	X
DosRmDir	√	√	√	√
DosScanEnv	√	√	√	X
DosSearchPath	√	√	√	X
DosSelectDisk	√	√	√	√
DosSetFHandState	√	√	√	√(R)
DosSetFileInfo	√	√	√	√
DosSetFileMode	√	√	√	√
DosSetFsInfo	√	√	√	√
DosSetMaxFH	√	√	√	X
DosSetVerify	√	√	√	√
DosWrite	√	√	√	√
DosWriteAsync	√	√	√	X
Errors and Exceptions				
DosErrClass	√	√	√	√
DosError	√	√	√	√(R)
DosSetVec	√	√	√	√(R)
Messages				
DosGetMessage	√	√	√	√(R)
DosInsMessage	√	√	√	√
DosPutMessage	√	√	√	√
Program Startup				
DosGetEnv	√	√	√	√
DosGetVersion	√	√	√	√
Code Page Support				
DosGetCp	√	√	√	√(R)
DosSetCp	√	√	√	√
DosSetProcCp	√	√	√	X
Country Support				
DosCaseMap	√	√	√	√
DosGetCollate	√	√	√	√
DosGetCtryInfo	√	√	√	√(R)
DosGetDBCSEV	√	√	√	√

Table E-2. Which Kbd Calls Where?				
Kbd Call	**PM**	**Vio-Windowed**	**Full-Screen**	**Family API**
KbdCharIn	X	√	√	√
KbdClose	X	√	√	X
KbdDeRegister	X	X	√	X
KbdFlushBuffer	X	√	√	√
KbdFreeFocus	X	√	√	X
KbdGetCP	X	√	√	X
KbdGetFocus	X	√	√	X
KbdGetStatus	X	√(R)	√	√
KbdOpen	X	√	√	X
KbdPeek	X	√	√	√(R)
KbdRegister	X	X	√	X
KbdSetCP	X	√	√	X
KbdSetCustXt	X	√	√	X
KbdSetFgnd	X	√(R)	√	X
KbdSetStatus	X	√(R)	√	√
KbdStringIn	X	√	√	√
KbdSynch	X	X	√	X
KbdXlate	X	√	√	X

Table E-3. Which Mou Calls Where?				
Mou Call	**PM**	**Vio-Windowed**	**Full-Screen**	**Family API**
MouClose	X	√(R)	√	X
MouDeRegister	X	X	√	X
MouDrawPtr	X	√(R)	√	X
MouFlushQue	X	√	√	X
MouGetDevStatus	X	√(R)	√	X
MouGetEventMask	X	√	√	X
MouGetNumButtons	X	√	√	X
MouGetNumMickeys	X	√	√	X
MouGetNumQueEl	X	√	√	X
MouGetPtrPos	X	√(R)	√	X
MouGetPtrShape	X	√(R)	√	X
MouGetScaleFact	X	√(R)	√	X
MouInitReal	X	X	√	X
MouOpen	X	√(R)	√	X
MouReadEventQue	X	√	√	X
MouRegister	X	X	√	X
MouRemovePtr	X	√(R)	√	X
MouSetDevStatus	X	√(R)	√	X
MouSetEventMask	X	√	√	X
MouSetPrtPos	X	√(R)	√	X
MouSetPtrShape	X	√(R)	√	X
MouSetScaleFact	X	√(R)	√	X
MouSynch	X	X	√	X

Table E-4 (Page 1 of 2). Which Vio Calls Where?				
Vio Call	**PM**	**Vio-Windowed**	**Full-Screen**	**Family API**
VioAssociate	√	X	X	X
VioCreateLogFont	√	X	X	X
VioCreatePS	√	X	X	X
VioDeleteSetId	√	X	X	X
VioDeRegister	X	X	√	X
VioDestroyPS	√	X	X	X
VioEndPopUp	√	√	√	X
VioGetAnsi	√	√	√	X
VioGetBuf	√	√	√	X
VioGetConfig	√	√	√	√
VioGetCp	√	√	√	X
VioGetCurPos	√	√	√	√
VioGetCurType	√	√	√	√
VioGetDeviceCellSize	√	X	X	X
VioGetFont	X	X	√	√
VioGetMode	X	√(R)	√	√
VioGetOrg	√	X	X	X
VioGetPhysBuf	X	X	√	√
VioGetState	X	X	√	√
VioModeUndo	X	X	√	X
VioModeWait	X	X	√	X
VioPopUp	√	√	√	X
VioPrtSc	X	X	X	X
VioPrtScToggle	X	X	X	X
VioQueryFonts	√	√	X	X
VioQuerySetIds	√	√	X	X
VioReadCellStr	√	√	√	√
VioReadCharStr	√	√	√	√
VioRegister	X	X	√	X
VioSavRedrawUndo	X	X	√	X
VioSavRedrawWait	X	X	√	X
VioScrLock	X	X	√	√(R)
VioScrollDn	√	√	√	√
VioScrollLf	√	√	√	√

Table E-4 (Page 2 of 2). Which Vio Calls Where?				
Vio Call	**PM**	**Vio-Windowed**	**Full-Screen**	**Family API**
VioScrollRt	√	√	√	√
VioScrollUp	√	√	√	√
VioScrUnLock	X	X	√	√(R)
VioSetAnsi	√	√	√	X
VioSetCp	√	√	√	X
VioSetCurPos	√	√	√	√
VioSetCurType	√	√	√	√
VioSetDeviceCellSize	√	X	X	X
VioSetFont	X	X	√	√
VioSetMode	X	√(R)	√	√
VioSetOrg	√	X	X	X
VioSetState	X	X	√	√
VioShowBuf	√	√	√	X
VioShowPS	√	X	X	X
VioWrtCellStr	√	√	√	√
VioWrtCharStr	√	√	√	√
VioWrtCharStrAtt	√	√	√	√
VioWrtNAttr	√	√	√	√
VioWrtNCell	√	√	√	√
VioWrtNChar	√	√	√	√
VioWrtTTY	√	√	√	√

Appendix F. PC-DOS Interrupts and Related OS/2 Function Calls

A PC-DOS application is written to run in a single-tasking environment and uses interrupts to request operating system functions. In some cases, interrupts can be easily mapped to OS/2 function calls. In others, functions are not as directly related.

The following tables list DOS and BIOS interrupts and function codes alongside related OS/2 function calls.

21H	DOS Function	OS/2 Call
00H	Program Terminate	DosExit
01H	Character input with echo	KbdStringIn
02H	Character output	VioWrtTTY
03H	Auxiliary input	DosOpen COMn DosRead COMn DosClose COMn
04H	Auxiliary output	DosOpen COMn DosWrite COMn DosClose COMn
05H	Printer output	DosOpen LPTn DosWrite LPTn DosClose LPTn
06H	Direct Console I/O	KbdCharIn VioWrtTTY
07H	Unfiltered character input without echo	KbdCharIn
08H	Character input without echo	KbdCharIn
09H	Output character string	VioWrtTTY
0AH	Buffered keyboard input	KbdStringIn
0BH	Get keyboard input status	KbdPeek
0CH	Reset keyboard buffer and input character	KbdFlushBuffer KbdCharIn
0DH	Reset the disk	DosBufReset
0EH	Set default disk	DosQCurDisk DosSelectDisk
0FH	Open a file	DosOpen
10H	Close a file	DosClose
11H	Find first matching file name	DosFindFirst

21H	DOS Function	OS/2 Call
12H	Find next matching file name	DosFindNext
13H	Delete a file	DosDelete
14H	Sequential read	DosRead
15H	Sequential write	DosWrite
16H	Create or truncate file	DosOpen
17H	Rename file	DosMove
19H	Current disk	DosQCurDisk
1BH	FAT information for default drive	DosQFsInfo
1CH	FAT information for specific drive	DosQFsInfo
21H	Random read	DosRead
22H	Random write	DosWrite
23H	Query file size	DosQFileInfo
24H	Set relative record field	DosChgFilePtr
25H	Set Interrupt vector	DosSetVec
27H	Random block read	DosRead
28H	Random block write	DosWrite
2AH	Get date	DosGetDateTime
2BH	Set date	DosSetDateTime
2CH	Get time	DosGetDateTime
2DH	Set time	DosSetDateTime
2EH	Set disk write verify switch	DosSetVerify
30H	Get system version number	DosGetVersion
35H	Get interrupt handler address	DosSetSigHandler
36H	Get disk free space	DosQFsInfo

21H	DOS Function	OS/2 Call
38H	Set or get country dependent information	DosGetCtryInfo
39H	Create a subdirectory	DosMkDir
3AH	Remove a subdirectory	DosRmDir
3BH	Change the current directory	DosChDir
3CH	Create a file	DosOpen
3DH	Open existing file	DosOpen
3EH	Close a file handle	DosClose
3FH	Read from a file or device	DosRead
40H	Write to a file or device	DosWrite
41H	Delete a file from a specified directory	DosDelete
42H	Move file read/write pointer	DosChgFilePtr
43H	Query or change file mode	DosQFileMode DosSetFileMode
44H	I/O control for devices	DosDevIOCtl
45H	Duplicate a file handle	DosDupHandle
46H	Force a duplicate of a file handle	DosDupHandle
47H	Get current directory	DosQCurDir
48H	Allocate memory block up to 64KB	DosAllocSeg
49H	Free allocated memory block	DosFreeSeg
4AH	Reallocate memory block up to 64KB	DosReallocSeg
4BH	Load or execute a program	DosExecPgm
4CH	Terminate a process	DosExit
4DH	Get return code of another process	DosCWait
4EH	Find first matching file	DosFindFirst
4FH	Find next matching file	DosFindNext

21H	DOS Function	OS/2 Call
54H	Get disk write verify setting	DosQVerify
56H	Rename a file	DosMove
57H	Get/set a file's date and time	DosQFileInfo DosSetFileInfo
59H	Get extended error	DosErrClass
5BH	Create a new file	DosOpen
5CH	Lock/unlock file access	DosFileLocks
62H	Get program segment prefix address	DosGetInfoSeg
65H	Get country information	DosGetCtryInfo
66H	Get/set code page	DosGetCp DosSetCp
67H	Set handle count	DosSetMaxFH
68H	Commit file to disk	DosBufReset
6CH	Extended open/create	DosOpen2

20H	DOS Function	OS/2 Call
00H	Terminate program	DosKillProcess

22H	DOS Function	OS/2 Call
00H	Terminate address	DosExitList

23H	DOS Function	OS/2 Call
00H	Ctrl-Break exit address	DosSetSigHandler

24H	DOS Function	OS/2 Call
00H	Critical error handler vector	DosError

25H	DOS Function	OS/2 Call
00H	Absolute disk read	DosDevIOCtl 8-64

26H	DOS Function	OS/2 Call
00H	Absolute disk write	DosDevIOCtl 8-44

00H	BIOS Function	OS/2 Call
00H	Divide by zero	DosSetVec

01H	BIOS Function	OS/2 Call
00H	Single step	DosPTrace

02H	BIOS Function	OS/2 Call
00H	Coprocessor exceptions	DosSetVec

03H	BIOS Function	OS/2 Call
00H	Breakpoint	DosPTrace

04H	BIOS Function	OS/2 Call
00H	Overflow	DosSetVec

05H	BIOS Function	OS/2 Call
00H	Print screen	VioPrtScreen

08H	BIOS Function	OS/2 Call
00H	System timer	DosTimerStart

09H	BIOS Function	OS/2 Call
00H	Lowest level keyboard intercept	DosMon calls

10H	BIOS Function	OS/2 Call
00H	Set video mode	VioSetMode
01H	Set cursor type	VioSetCurType
02H	Set cursor position	VioSetCurPos
03H	Get cursor position	VioGetCurPos
06H	Scroll up	VioScrollUp
07H	ScrollDown	VioScrollDn
08H	Read character and attribute	VioReadCellStr
09H	Write character and attribute	VioWrtCellStr
0AH	Replicate a character	VioWrtNChar
0BH	Set palette registers	VioSetState
0EH	Write teletype mode	VioWrtTTY

10H	BIOS Function	OS/2 Call
0FH	Get video mode	VioGetMode
10H	Set palette registers	VioSetState
11H	Load a font	VioSetFont
12H	Get configuration information	VioGetConfig
13H	Write a character string	VioWrtCharStr
1AH	Get display adapter type	VioGetConfig
1BH	Get state information	VioGetState
1CH	Save the screen context	VioModeWait

11H	BIOS FUNCTION	OS/2 Call
00H	Equipment determination	DosDevConfig

12H	BIOS Function	OS/2 Call
00H	Memory size	DosMemAvail

13H	BIOS Function	OS/2 Call
02H	Read sectors from disk or diskette	DosDevIOCtl 9-64
04H	Verify sectors	DosDevIOCtl 9-65
07H	Format drive starting at desired cylinder	DosDevIOCtl 9-45
08H	Read drive parameters	DosDevIOCtl 9-63

14H	BIOS Function	OS/2 Call
00H	Initialize COM port	DosDevIOCtl 1-41 DosDevIOCtl 1-42 DosDevIOCtl 1-66 DosDevIOCtl 1-67 DosDevIOCtl 1-6D DosDevIOCtl 1-72
01H	Write characters to COMn	DosDevIOCtl 1-44
03H	Get status of COM port	DosDevIOCtl 1-66 DosDevIOCtl 1-67 DosDevIOCtl 1-6D DosDevIOCtl 1-72

14H	BIOS Function	OS/2 Call
03H	Extended initialization	DosDevIOCtl 1-41 DosDevIOCtl 1-42 DosDevIOCtl 1-66 DosDevIOCtl 1-67 DosDevIOCtl 1-6D DosDevIOCtl 1-72
05H	Extended communications port control	DosDevIOCtl 1-46

15H	BIOS Function	OS/2 Call
4FH	Keyboard intercept	Keyboard monitors
83H	Event wait	DosSetSigHandler
86H	Wait	DosSleep
C2H	Pointing device interface	MouXXX API

16H	BIOS Function	OS/2 Call
00H	Input character	KbdCharIn
01H	Examine data record	KbdPeek
02H	Get shift key state	KbdGetStatus
05H	Keyboard write	Keboard Monitor
10H	Input character	KbdCharIn
11H	Examine data record	KbdPeek
12H	Get keyboard state	KbdGetStatus

17H	BIOS Function	OS/2 Call
00	Output to printer	DosOpenLPTn DosWrite LPTn DosClose LPTn
01	Initialize LPT port	DosDevIOCtl 5-46
02	Get status of LPT port	DosDevIOCtl 5-66

1AH	BIOS Function	OS/2 Call
02H	Read real-time clock time	DosGetDateTime
03H	Set real-time clock time	DosSetDateTime
04H	Read real-time clock date	DosGetDateTime
05H	Set real-time clock date	DosSetDateTime

1AH	BIOS Function	OS/2 Call
06H	Start asynchronous timer	DosTimerAsync
07H	Stop timer	DosTimerStop

Appendix G. System Limits

This appendix contains tables of system limits for:

- Memory resources
- Sessions
- Presentation Manager resources
- Hardware
- Various system resources.

You should be aware of these limits if you program large applications or applications that work with other applications.

Generally, system limits are high enough so that no single application will reach them, providing the user has sufficient disk space and system memory.

The limits for individual applications listed in the tables are maximum amounts available.

Recovery Operations

Here are some recommended responses you should provide to users when the system reaches a limit:

- If a limit is reached when the user tries to start an application, the system tells the user about the problem. The user must then take appropriate action.
- If a limit is reached when the user requests an action, you should tell the user about the problem and restore the application to the condition it was in before the limit was reached.

Memory Resource Limits

Memory Resource	System Limit	Available to Applications
Segment size	64KB	64KB
Segments	50 967	50153
Swappable segments	16 000	15 588
Discardable segments	Available memory	250 (number loaded by OS/2)
Shared segments per process	6 144	—
Non-shared segments per process	2 048	—
Fixed segments	Available memory	53 (number loaded by OS/2)
Named shared segments per process	30	30
Libraries	Available memory	29 (number loaded by OS/2)
Global descriptor tables	1	1
Global descriptor table entries	8 192	6 997
Local Descriptor Tables	1	1
Local Descriptor Table entries	8 192	8 192

Session Limits

Session Type	System Limit	Available to Applications
Full-screen	16	12
Presentation Manager	16	14
Text-window	16	16

Presentation Manager Limits

Presentation Manager Resource	System Limit
String atoms	16 384
Integer atoms	49 152
Heap size (1 segment)	64KB
Segments in segment store of presentation space	16 378
Logical fonts per presentation space	254
Size of logical color table	31KB
Presentation space (1 segment)	64KB
Presentation spaces per process	32
Horizontal scan lines per cursor	32
Cursor width (text mode)	1

Hardware Limits

Hardware Part	System Limit
Fixed disks	7
Disk partition size	2GB
Logical drives	26
Diskette drives	3
Parallel ports	3
Serial ports	3
Displays	1
Mice	1
Keyboards	1
Compatibility boxes	1

Limits for Various Operating System Resources

Resource	System Limit	Available to Applications
Processes	511	504
Threads	511	483
Threads per process	53	53
Open file handles	64 000	—
Pipes	Part of open file handles	Part of open file handles
Pipe size	64KB	—
Queues	1 448	1 256
Queue priorities	16	—
Queue elements per queue	3 268	—
System semaphores	256	213
Semaphores with outstanding waits	128	126
Maximum semaphores per DosMuxSemWait	16	—
System signals per process	3	—
User signals per process	3	—

Threads:

The minimum available space on the stack for a thread calling OS/2 function is 4KB. The system allows only one timer per thread.

Appendix H. Information Presentation Facility Tags

This appendix discusses the tags you use to mark up files to create helps.

Most tags start with a colon and end with a period. Some tags require an end tag to indicate explicitly that the tag has ended. End tags have the end tag delimiter (:e) followed by the name of the tag it is ending and a period (.). For example,

```
:userdoc.
:euserdoc.
```

A few tags, called control words, begin with a period and have no accompanying end control word. These control words provide additional instructions about how your file is compiled. In the following example, the imbed control word instructs the compiler to include the file named *newfile* at compile time.

```
.im newfile
```

The Information Presentation Facility tags are listed in alphabetical order. The attributes and elements required to support each tag are listed in the syntax layout for each tag. Attributes followed by apostrophes or single quotation marks show that the information needed contains special characters and requires single quotation or apostrophe delimiters.

Certain tags are needed and are required to be in a specific order before the file can be compiled. The following layout shows the minimum tags required to compile a file:

```
:userdoc.
:h1 id=test1.How to
:p.
This describes how to do something.
:p.
More text.
:euserdoc.
```

.* (Comment)

Purpose
The comment control word is used to place comments in a file.

Syntax

Tag	Element	Attributes	End
*	Comment		

Attributes
None

Description
Comments can be used to reference items, to place notes in your files, or to prevent the printing of an item. The comment control word is the first tag on the line of text that you do not want displayed. Text on the same line as the comment control word is not printed when the file is compiled. You can have as many comments in your file as you need; however, each comment control word must begin on a new line and must be the first tag on the line. No space is required between the comment tag and the text that follows it. Comment tags are used independently of other tags, that is, they are not used between any other tags or with any other tag and its accompanying text. The comment control word has no attributes and requires no end tag.

Conditions
Do not use the comment control tag:

- Within tags, that is between the colon that starts the tag and the period that ends the tag.
- Between a tag and its accompanying text.

Example

```
.* Comment control words should be at the start of the line.
.* When the file is compiled, the information on the comment
.* line is not printed.
```

Output
When the file is compiled, the comment tag and the information following it is not printed.

:artlink. (Artlink)

Purpose
The artlink tags define the link definition file for hypergraphic links.

Syntax

Tag	Element	Attributes	End
:artlink.			:eartlink.

Attributes
None

Description
The artlink tags are similar to the userdoc tags in that they define a file type. The artlink tag must be the first tag in the link definition file. The artlink tags are valid only when used with the artwork tag to describe the graphic links identified in the link definition file. The *linkfile=' '* attribute of the artwork tag identifies the link file as follows: *linkfile='artfilename.xxx'*. See the "Artwork" tag for information about the linkfile attribute.

The link tags within an artlink description are used to specify data links required to provide hypergraphic links from a bit map. Link information is placed between the artlink tag and the end artlink tag. The data for the link information is identified with the link tag and follows the link tag format. The exception is that when the link tag is used within the link definition file, an end link tag is not required.

The link attributes required for each link depend on the type of link you are providing.

When linking from a whole bit map, use one link tag to identify the bit map location. Values identifying the link location of the hypergraphic area are not required. Example 1 provides an example of how you could enter the linking data in the link definition file.

Example 2 provides an example of how you could enter the data for linking from a segmented bit map in the link definition file.

When linking from a segmented bit map, use a link tag to identify each segment or *hypergraphic* area of the bit map. You must also include the values which identify the link locations. The *hypergraphic area* is the boundary that defines the horizontal and vertical areas of the bit map segments. These segments are identified by the values specified for x, y, cx, and cy. The beginning bit map segment provides the values for x and y. The values for cx and cy identify the change from the beginning values. A segmented bit map could be defined like this.

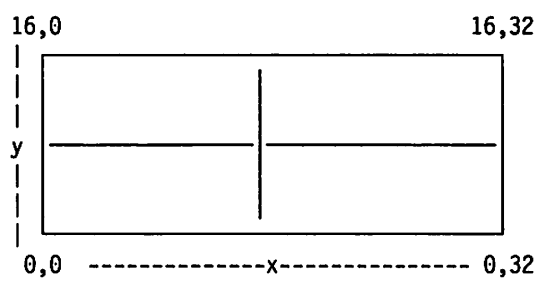


Figure H-1. Segmented bit map

On bit maps, the x axis is always horizontal and the y axis is always vertical. The values 0,0, indicating the beginning of the bit map, are always at the bottom left corner of the bit map. See the "Using the Graphics Programming Interface" section for more information about bit maps.

Conditions

- The artlink tags are valid only in the file identified by the *linkfile=' '* attribute.
- When linking to a segmented bit map, a link tag and the reftype are required. Other required attributes depend on the type of link you are providing.
- When linking from a whole bit map only one link tag is required.

Example

Example 1. This example shows how to tag when the link is from a whole bit map. The name of the whole bit map file is *whbfile.bmp*. The file containing the link information is **whflink.art**.

```
:artwork name='whbfile.bmp' linkfile='whflink.art'.
```

The following information is placed in the **whflink.art** link definition file.

```
:artlink.
:link refid=fmt321 reftype=hd.
:eartlink.
```

Example 2. This example shows how to tag when the link is from a segmented bit map. The name of the segmented bit map is **show2.bmp**. The name of the file containing the link information is **link.dat**.

```
:artwork name='show2.bmp' linkfile='link.dat'.
```

The following information could be placed in the **link.dat** link definition file.

```
:artlink.
:link reftype=hd res=001 x=0 y=0 cx=16 cy=8.
:link reftype=fn refid=afnr x=16 y=8 cx=16 cy=8.
:link reftype=inform res=0345 x=0 y=8 cx=16 cy=8.
:link reftype=launch object='c:\os2\e.exe' data='c:\appsdir\tutor.dat'
       x=16 y=0 cx=16 cy=8.
:eartlink.
```

Output

Example 1. In this example, the link is from a whole bit map file. The area defining the bit map is not required.

Example 2. In this example, the link is from a segmented bit map. The hypergraphic area of each bit map segment is required to be included in the link definitions.

:artwork. (Artwork)

Purpose

The artwork tag identifies a bit map to be placed in the user file.

Syntax

Tag	Element	Attributes	End
:artwork.	Artwork	name	
		align	
		linkfile	
		runin	

Attributes

name = *'filename.ext'*

This is the name of the artwork file. The name should include the complete name of the file, including the extension. The name of the artwork file is required for all artwork.

align = **left** | **right** | **center**

Align specifies how the graphic is to align with the current margin. The alignment can be to the left on the panel, to the right on the panel, or centered on the panel.

linkfile = *'filename.ext'*

This attribute names the link definition file and can be omitted if the artwork file does not require links. The link definition file contains the link information. The name of the file must include the complete filename, including the extension. The file named by the linkfile attribute begins with the artlink tag and ends with the end artlink tag. Link tags and the accompanying attributes that specify the links from the bit map follow the link tag format and are placed between the artlink tags. See the "Link" tag for more information about the link tag format.

runin

Specifies that you want the artwork placed within the line of text. Tag the file by entering the artwork tag and its attributes in the line of text where you want the artwork to appear.

Description

Use the artwork tag to include in the text file any Presentation Manager bit maps created on external media such as vectors, scanned images, or graphic/image tools. The attributes accompanying the artwork tag allow you to merge whole or segmented art and to position the art in the help panel. The name attribute is required and must include a fully qualified file name.

The artwork tag and the *linkfile* = ' ' attribute allow you to link from whole or segmented bit maps. The file identified by the linkfile attribute contains the data for linking from the bit map. This file begins with the artlink tag and ends with the end artlink tag. The link tag is used to specify the link information. Link information includes all parameters and values required to link from the bit map.

A whole or segmented bit map can be created by an application or it can be created using the Presentation Manager Icon Editor. When defining links from a segmented bit map, you must include the values specified for each segment or hypergraphic area of the bit map. The *hypergraphic area* is the boundary that defines the horizontal and vertical areas of the bit map segments. The segments are identified by the values provided for x, y, cx, and cy. The beginning values are specified by x and y. The values for cx and cy identify the change from the beginning values. A segmented bit map could be defined like this.

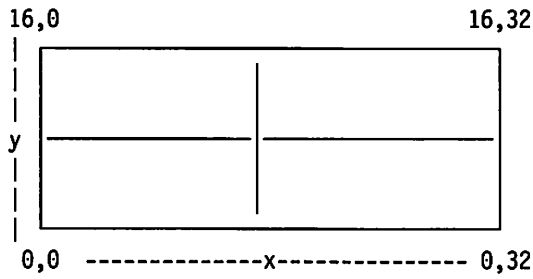


Figure H-2. Segmented bit map

The x axis is always horizontal and the y axis is always vertical. The values 0,0, indicating the beginning of the bit map, are always at the bottom left corner of the bit map. See the "Using the Graphics Programming Interface" section for more information about bit maps. When linking from a segmented bit map, the hypergraphic area values specified by x, y, cx, and cy are required to identify the specific location of the link. When the link is from a whole bit map, the values for the hypergraphic area can be omitted.

Conditions

- A full file name, including the extension is required.
- The linkfile attribute names the link definition file.
- The artwork tag must be accompanied by the name attribute.

Example

Example 1. This example shows how to include artwork that does not require a hypergraphic link.

```
:artwork name=gopi.art runin.
```

Example 2. This example shows how to tag a file when the link is from a whole bit map. The artwork tag names the bit map file; the *linkfile*=' ' attribute names the link definition file.

```
:artwork name='wholefle.bmp.' runin linkfile='wholelnk.art'.
```

To tag the link definition file, you could enter:

```
:artlink.
:link res=321 reftype=hd.
:eartlink.
```

Example 3. This example shows how to tag to link from a hypergraphic. The name of the segmented bit map file is **show2.bmp**. The name of the file containing the link information is **link.dat**. When including the segmented bit map, your markup could look like this:

```
:artwork name='show2.bmp' runin linkfile='link.dat'.
```

To tag the link definition file, you could enter:

```
:artlink.
:link reftype=hd res=001 x=0 y=0 cx=16 cy=8.
:link reftype=fn refid=ftntid x=16 y=8 cx=16 cy=8.
:link reftype=inform res=0345 x=0 y=8 cx=16 cy=8.
:link reftype=launch object='c:\os2\e.exe' data='c:\appsdir\tutor.dat'
       x=16 y=0 cx=16 cy=8.
:eartlink.
```

Output

Here is an explanation of the link data information in example 3. The hypergraphic area for each link is identified by the values specified for x, y, and cx, and cy. See the "Using the Graphics Programming Interface" section for more information about bit maps.

- In the first line, the reftype is a head; the link resource identifier *res=001*, identifies the head.
- In the second line, the reftype is a footnote; the *refid=fntnt* references the identifier for the footnote.
- In the third line, the reftype is inform. The *res=0345* identifies the message. The compiler sends an HM_INFORM message with the *res=0345* identifier to the application. The action taken by the application is predetermined in the application's code.
- In the fourth line, the reftype launch provides a link to another program. The object attribute includes a fully qualified file name, which identifies the program you want to start.

:caution. (Caution)

Purpose
Use the caution tags to alert the user to a risk.

Syntax

Tag	Element	Attributes	End
:caution.	Caution	text	:ecaution.

Attributes
text=' '
This attribute allows you to change the word *CAUTION* to different text.

Description
A caution message should appear before the text to which it pertains so that the user will read about the possible risks before reading the text.

When a caution is displayed, the word *CAUTION* appears on the screen and the caution text is displayed on the next line. A blank line is printed before the tagged area when the file is displayed.

You can use the *text=* attribute to replace *CAUTION* with other text if you prefer.

Conditions
None

Example
```
:caution.
These berries are wild.  Do not eat.
:ecaution.
```

To provide a specific caution, you could enter:

```
:caution text='Wild Berries:'.
These berries are wild.  Do not eat.
:ecaution.
```

Output
CAUTION:
These berries are wild. Do not eat.

Wild Berries:
These berries are wild. Do not eat.

:cgraphic. (Character graphic)

Purpose
Use the cgraphic tag to define an area that has a character graphic.

Syntax

Tag	Element	Attributes	End
:cgraphic.	Character graphic		:ecgraphic.

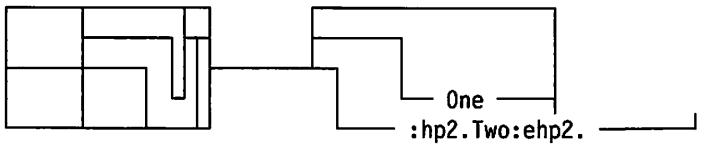
Attributes
None

Description
Character graphics are graphics you create with an ASCII editor. The character graphic tag format is similar to an example tag format. Both tags instruct the compiler to use a monospace font to format the data. The difference is the example tag instructs the compiler to use a special font to format text, and the character graphic tag instructs the compiler to use special characters to create the drawings. The character graphic tag defines the area that has character graphics and causes a blank line before and after the tagged area. The character graphic tag has no attributes.

Conditions
None

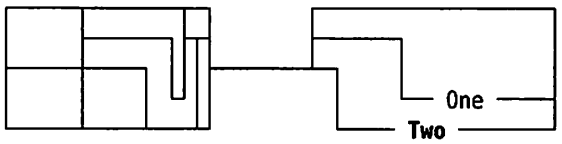
Example

```
:cgraphic.
```




```
:ecgraphic.
```

Output



:color. (Color)

Purpose
The color tag is used to change the color and background of the text.

Syntax

Tag	Element	Attributes	End
:color.	Color	fc bc	

Attributes

fc = default | blue | cyan | green | neutral | red | yellow
The foreground color attribute allows you to change the color of text. Text following this attribute displays in the color specified.

bc = default | blue | cyan | green | neutral | red | yellow
The background color attribute allows you to change the background color of the text. The screen colors remain the same.

Enter `Default` as the foreground or background color when you want to return to the colors originally set for your system.

Description
The color tag allows you to change the color of the text and of the background of the text. The color tag is similar to highlight phrase color tags (hp4, hp8, and hp9). The difference is that highlight phrase color tags set the color of the text while the color tag can set the color of the text and the color of the background. The attributes specify whether you are setting the color of the text (foreground) or of the background for the text (background). Colors set with the color tag remain until another color is specified.

Conditions
None

Example

```
:p.
Use the :color bc=cyan fc=red.
tag to change the color of the
text and of the screen. :color fc=green.
Colors can be used to
highlight text.
```

Output
The colors of the screen and text change to the specified colors.

:dd. (Definition description)

Purpose
The definition description tag identifies the term being defined in a definition list.

Syntax

Tag	Element	Attributes	End
:dd.	Definition description		

Attributes
None

Description
The definition description tag defines the word specified with the definition term tag. The term appears in the left column and the text identified by this tag appears in the right column.

The attributes specified on the definition list tag determine the column size and the way the term and description format. See the "Definition list" tag for more information. The definition description tag is paired with the definition term tag. When entering text, the definition description tag follows the definition term tag. You can enter multiple definition description tags within a definition list. However, each definition description tag requires an accompanying definition term tag. The definition description tag has no attributes and requires no end tag.

Conditions
- Each definition description tag must be paired with a definition term tag.
- The tags are valid only within a definition list.

Example
```
:dl compact tsize=10.
:dthd.Item
:ddhd.Description
:dt.Tree
:dd.Plant life in forest
:dt.Orange
:dd.Fruit on tree
:dt.Cow
:dd.Animal on farm
⋮
:edl.
```

Output

Item	Description
Tree	Plant life in forest
Orange	Fruit on tree
Cow	Animal on farm

:ddhd. (Definition description head)

Purpose

The definition description head tag identifies the heading for the description column in a definition list.

Syntax

Tag	Element	Attributes	End
:ddhd.	Definition description head		

Attributes

None

Description

The definition description head tag is paired with the definition term head tag. Both tags are only valid within a definition list. When tagging a file, the definition term head tag precedes the definition description head tag. When the file is displayed, the definition heading appears above the text, in the same column and font as the text. The definition description head tag is optional. It has no attributes and does not require an end tag.

Conditions

- The definition description head tag is valid only within a definition list.
- The definition description head must be paired with a definition term head tag.

Example

```
:dl compact tsize=10.
:dthd.Item
:ddhd.Description
:dt.Tree
:dd.Plant life in forest
:dt.Orange
:dd.Fruit on tree
:dt.Cow
:dd.Animal on farm
:
:edl.
```

Output

Item	Description
Tree	Plant life in forest
Orange	Fruit on tree
Cow	Animal on farm

:dl. (Definition list)

Purpose

The definition list tags identify a list of words and their definitions.

Syntax

Tag	Element	Attributes	End
:dl.	Definition list	tsize compact break	:edl.

Attributes

tsize = 10 | n

Specifies the space to be allocated for the definition list terms and term headings. The default is 10 character units. If the tsize value exceeds the current size of the formatting area (the space between the current left and right margins), the tsize is assigned the current formatting area size and a warning message is issued.

compact

The *compact* attribute causes the list to format without a blank line between the items.

break = none| fit | all

Controls the formatting of the definition terms and descriptions.

break = none

The definition description is on the same line as the definition term. If the length of the term exceeds the specified tsize value, the term spills over into the description column and the definition description starts one space after the term.

break = fit

The definition description is placed on the line below the term if the term is longer than the tsize value.

break = all

All descriptions are placed on the line below the term.

Description

Use the definition term tag and the definition description tag to identify the terms you are defining and their descriptions. Use the definition term head and the definition description head tags to define headings for the terms and descriptions. Heading are optional. When the heading tags are used, they are entered after the definition list tag and before the definition term and definition description tags.

You can nest definition lists within other definition lists and include specific values for each list. If you do not set a tsize, the space allocated for the term size is 10 character units. When definition lists are displayed, a blank line precedes the tagged area. If you do not specify a break value, the text for the description formats on the same line as the term.

Conditions

- The definition term head tag is paired with the definition head description tag and precedes the definition term and definition description tags.
- The definition term tag requires a definition description tag.

Examples

```
:dl compact tsize=10.
:dthd.Item
:ddhd.Description
:dt.Tree
:dd.Plant life in forest
:dt.Orange
:dd.Fruit on tree
:dt.Cow
:dd.Animal on farm
⋮
:edl.
```

Output

Item	Description
Tree	Plant life in forest
Orange	Fruit on tree
Cow	Animal on farm

:dt. (Definition term)

Purpose
The definition term tag identifies the term being defined in the definition list.

Syntax

Tag	Element	Attributes	End
:dt.	Definition term		

Attributes
None

Description
The definition term tag is paired with the definition description tag. The definition term tag precedes the definition description tag. The definition term tag appears in the left column, and the associated definition description tag appears in the right column.

The attributes specified on the definition list tag determine the column size and the way the term and description format. If the tsize attribute is smaller than the definition term and the break attribute is set to none, the definition term spills over into the definition description column. See the "Definition list" tag for more information. You can enter multiple definition term tags within a definition list; however, each definition term tag requires an accompanying definition description tag. The definition term tag has no attributes and requires no end tag.

Conditions
- The definition term tag is valid only within a definition list.
- The definition term tag requires a definition description tag.

Example
```
:dl compact tsize=10.
:dthd.Item
:ddhd.Description
:dt.Tree
:dd.Plant life in forest
:dt.Orange
:dd.Fruit on tree
:dt.Cow
:dd.Animal on farm
⋮
:edl.
```

Output

Item	Description
Tree	Plant life in forest
Orange	Fruit on tree
Cow	Animal on farm

:dthd. (Definition term head)

Purpose
The definition term head tag identifies the heading for the term column in a definition list.

Syntax

Tag	Element	Attributes	End
:dthd.	Definition term head		

Attributes
None

Description
The definition term head tag is paired with the definition description head tag. Both tags are valid only within a definition list. When tagging a file, the definition term head tag precedes the definition description head tag. When the file is displayed, the definition heading appears above the text, in the same column and font as the text it describes. The definition term head tag is optional.

Conditions
- The definition term heads are valid only within a definition list.
- Each definition term head tag requires a definition description tag.

Example
```
:dl compact.
:dthd.Item
:ddhd.Description
:dt.Tree
:dd.Plant life in forest
:dt.Orange
:dd.Fruit on tree
:dt.Cow
:dd.Animal on farm
:
:edl.
```

Output

Item	Description
Tree	Plant life in forest
Orange	Fruit on tree
Cow	Animal on farm

:fig. (Figure)

Purpose
The figure tags are used to identify a figure.

Syntax

Tag	Element	Attributes	End
:fig.	Any text		:efig.

Attributes
None

Description
The figure tag specifies that the information following the tag be formatted as it is entered. Lines of text that exceed the panel area are clipped. The figure is displayed in proportional font, and a blank line precedes the text. Because the figure tag specifies proportional font, words line up but letters and numbers may not.

Conditions
None

Example:

```
:fig.
 Area   Number  Classification         Code

|GLWEB11
|                                              |
|       2      Full-time exempt        1A      |
|       4      Part-time exempt        1B      |
|       4      Full-time non-exempt    2A      |
|       1      Part-time non-exempt    2B      |
|       2      Supplemental            3A      |
|                                              |
|______________________________________________|

:efig.
```

Output

```
 Area   Number  Classification         Code

|GLWEB11
|                                              |
|       2      Full-time exempt        1A      |
|       4      Part-time exempt        1B      |
|       4      Full-time non-exempt    2A      |
|       1      Part-time non-exempt    2B      |
|       2      Supplemental            3A      |
|                                              |
|______________________________________________|
```

:figcap. (Figure caption)

Purpose
The figure caption tag provides a title for the figure.

Syntax

Tag	Element	Attributes	End
:figcap.	Any text		

Attributes
None

Description
The figure caption tag is placed between the figure and figure end tag. The text of the caption goes on the same line as the figure caption tag or on the next line. The figure caption tag requires no end tag.

Conditions
- Use the figure caption tag either just after the figure tag or just before the end figure tag.
- The figure caption text cannot contain tags or semicolons.

Example

```
:fig.
 Area   Number  Classification         Code

|CDPG1
|                                              |
|       5      Full-time exempt        1A      |
|       1      Part-time exempt        1B      |
|       3      Full-time non-exempt    2A      |
|       1      Part-time non-exempt    2B      |
|       1      Supplemental            3A      |
|                                              |
|______________________________________________|

:figcap.Payroll Codes for Area CDPG1
:efig.
```

Output

Area	Number	Classification	Code
CDPG1			
	5	Full-time exempt	1A
	1	Part-time exempt	1B
	3	Full-time non-exempt	2A
	1	Part-time non-exempt	2B
	1	Supplemental	3A

Payroll Codes for Area CDPG1

:fn. (Footnote)

Purpose
The footnote tags identify a pop-up window of information.

Syntax

Tag	Element	Attributes	End
:fn.	Pop-up	id	:efn.

Attributes

id =

The footnote tag uses an ID attribute to reference the information. The *reftype=* attribute is not a footnote attribute; however, it is required when defining the link to be a footnote. See the "Link" tag for more information.

Description
The footnote tags create a cross-reference to information that is displayed in a pop-up window within the panel. The footnote tag requires the *id=* attribute. You can include multiple footnotes, however you must end one footnote before beginning another one. Index entries are not valid within a footnote. Footnotes can appear within paragraphs, lists, highlighted phrases, and artwork.

Conditions

- Index entries cannot appear in a footnote.
- The footnote tag requires an *id=* attribute.
- One footnote must end before another footnote begins.

Example
The following example shows how to enter the footnote ID and how to provide a link to the footnote. In the following example the footnote ID is *drive*. To link to a footnote enter:

```
:fn id=drive.
The information you place here appears in the pop-up window as a
footnote.  For example, you could enter additional information
about the disk drive in a footnote. :efn.
```

To provide the link that allows the user to view the footnote panel, you could enter:

```
:p.Additional information about
:link refid=drive reftype=fn.disk drives :elink.
is available.
```

Output
When the file is displayed, *disk drives* is highlighted and clicking on *disk drives* pops up the footnote window containing the additional information.

:h1. through :h6. (Headings)

Purpose

The heading tags identify main topics and subtopics of information.

Syntax

Tag	Element	Attributes	End
:h1.	Heading level 1	id res name hide tutorial	
:h2.	Heading level 2	id res name hide tutorial	
:h3.	Heading level 3	id res name hide tutorial	
:h4.	Heading level 4		
:h5.	Heading level 5		
:h6.	Heading level 6		

Attributes

id=
This attribute is used to specify an identifier for the heading. The ID can consist of alpha or alphameric characters.

res=
This attribute is a resource identifier. It can be any integer ranging from 1 to 32000. When linking files, the integer for the link information must match the integer of the tag it is being linked to.

name=
This attribute is the same as the *id=* attribute. You can use either alpha characters or alphameric characters to specify the name of the identifier.

hide
This attribute prevents the head from displaying in the table of contents. The hidden head and the information can be viewed by providing a link to the head.

tutorial=
This attribute specifies the name of the tutorial which is defined by the application and is associated with a panel. This attribute causes *Tutorial* to be displayed as a choice in the Main Help window Help pull-down. When *Tutorial* is selected, an HM_TUTORIAL message is sent to the application with the tutorial name specified with this attribute.

Description

Headings group the information into logical sections and can contain up to 70 characters, including spaces. If the heading information exceeds the panel width, the words are wrapped. The head levels show the hierarchy of the information you present and its importance to the topic. For example, a head 2 follows a head 1. You cannot skip head levels. For example, you cannot go from a head 3 to a head 6. Headings also supply the table of contents entries. The compiler automatically places headings 1, 2 and 3 in the table of contents. Other headings (headings 4, 5 and 6), appear on the specified panel. When the file is displayed, there is no visual difference between these head levels.

Attributes for a heading can be *id=*, *name=*, or *res=*. The resource identifier is recommended.

The *hide* attribute can be used with any heading identifier *(name=, id=, or res=)* to prevent a heading from displaying in the table of contents. The head and text can be seen only if there is a link from the head. The user sees this information only if a link is provided from the head.

The *tutorial=* attribute and an accompanying identifier is used to create a tutorial help, which is displayed as the last item in the Main Help window Help pull-down.

The head level tags require no end tag.

Conditions

- Head levels follow in numerical sequence. A head level 2 follows a head level 1. You cannot skip from a head level 3 to a head level 6.

Example

Example 1. This example shows how to tag headings, using each attribute.

```
Example 1
:h1 res=150.Copy
⋮
:h2 res=151 id=fle.Files
⋮
:h3 res=152.File Names
:p.Special keys are required...
:h1 res=153 hide.Keys for Hidden Text
:h2 res=154 tutorial=tags.
```

Example 2. This example shows how to provide a link from the RES for head 1.

```
Example 2

Click on
:link reftype=hd res=150. Copy format :elink.
for additional information on how to copy files.
```

Example 3. This example shows how to link from the ID for head 2.

```
Example 3

Click on :link reftype=hd id=fle. files :elink.
for information on how to set up and maintain files.
```

Example 4. This example shows how to provide a link from the hidden head.

```
Example 4

Special :link reftype=hd res=154. keys :elink.
are required...
```

Output

Example 1. Example 1 shows how to tag the file to allow hypertext links.

Example 2. When the file is displayed, *Copy format* is highlighted, indicating that it is a hypertext phrase. Clicking on *Copy format* links to the head *Copy* and the information on that panel is displayed.

Example 3. When the file is displayed, *files* is highlighted, indicating that it is a hypertext phrase. Clicking on *files* links to the head with *id=fle* and the information on that panel is displayed.

Example 4. When the file is displayed, *keys* is highlighted, indicating that it is a hypertext phrase. Clicking on *keys* links to the head with *res=154*. The hidden panel and the information on that panel is displayed.

:hide. (Hide)

Purpose
The hide tag is used to allow conditional viewing of specific text.

Syntax

Tag	Element	Attributes	End
:hide.	Hide	key	:ehide.

Attributes

key='name(n)'

This attribute allows you to set specific keys to allow different users to view provisory display of information. You can set one key name or multiple key names. When entering multiple key names, use the plus (+) sign after each name and leave a space between each name. Enclose the names within the string within apostrophes. Text entered between the hide and end hide tag can be viewed only when the *key='name'* attributes matches the entry specified by the user. Use the OS/2 environment variable *Set Keys=* to specify the key name identified for the hidden information.

Description
The hide tag can be used to add security to the file by allowing access to specific users while disallowing access to users who are not qualified to view the specified information. The designated users can access the information by setting the *Keys=* variable at the OS/2 command prompt to match the key= attribute set for the hidden information. The environment variable can also be set when the system is installed. The value set at the OS/2 command prompt or when the system is installed has to match the *key='name'* attribute set for hidden information. If there is no match, the hidden information is not displayed.

When tagging the file, you can include multiple hide tags within a panel. Each hide tag can have the same or a different *key='name'* attribute. However, to access the hidden information the user must be aware of the key name specified for the hidden data.

Conditions

- One hide tag must end before another hide tag begins.

Example

```
To department J12
:p.
The following changes are effective immediately.
Gladys and Pete are testing all new widgets.
:hide key='gl+pc'.
The defect rate of the new shipment of widgets is 40%.
Determine if this high rate is because of the widgets or because of
the recent change in equipment.
:ehide.
Scott is responding to and bringing to closure
all open items.
:hide key='sak'.
Open item    Status
axc0056      Meet with LG Friday, 4:30, your office.
axc0297      BJ is working to close; may need your input.
axc0419      Pending feedback from Rick; due Tuesday.
axc9044      Opened by management today, see Tracy for details.
:ehide.
```

Output

Example 1. This information is viewed by all department members.

```
To department J12

The following changes are effective immediately.
Gladys and Pete are testing all new widgets.
Scott is responding to and bringing to closure
all open items.
```

Example 2. When the key name *gl+pc* is set in the environment variable, the following information is displayed for the owners of that key name.

```
To department J12

The following changes are effective immediately.
Gladys and Pete are testing all new widgets.
The defect rate of the new shipment of widgets is 40%.
Determine if this high rate is because of the widgets or because of
the recent change in equipment.
Scott is responding to and bringing to closure
all open items.
```

Example 3. When the key name *sak* is set in the environment variable the following information displays for the owners of that key name.

```
To department J12

The following changes are effective immediately.
Gladys and Pete are testing all new widgets.
Scott is responding to and bringing to closure
all open items.
Open items   Status
axc0056      Meet with LG Friday, 4:30, your office.
axc0297      BJ is working to close; may need your input.
axc0419      Pending feedback from Rick; due Tuesday.
axc9044      Opened by management today, see Tracy for details.
```

In each example, notice that the information is hidden from users who do not have access to the key name specified for the hidden information.

:hp. (Highlight phrases)

Purpose

Highlight phrase tags allow you to highlight text, to change the text font, and to add color to the text. The highlight phrase tags are listed in the following table.

Syntax

Tag	Element	Results	End
:hp1.	Highlighted phrase level 1	*Highlight phrase 1 looks like this.*	:ehp1.
:hp2.	Highlighted phrase level 2	**Highlight phrase 2 looks like this.**	:ehp2.
:hp3.	Highlighted phrase level 3	***Highlight phrase 3 looks like this.***	:ehp3.
:hp4.	Highlighted phrase level 4	Color 1 (Red)	:ehp4.
:hp5.	Highlighted phrase level 5	Highlight phrase 5 looks like this.	:ehp5.
:hp6.	Highlighted phrase level 6	*Highlight phrase 6 looks like this.*	:ehp6.
:hp7.	Highlighted phrase level 7	**Highlight phrase 7 looks like this.**	:ehp7.
:hp8.	Highlighted phrase level 8	Color 2 (Blue)	:ehp8.
:hp9.	Highlighted phrase level 9	Color 3 (Pink)	:ehp9.

Attributes

None

Description

Use the highlighting phrase tags to highlight words and phrases to change the font of the text, or to change the color of the text. Highlight phrase tags require an end highlight phrase tag and have no attributes.

Conditions

- One highlight phrase must end before another one begins.

Example

```
Read :hp7.all:ehp7. instructions
:hp2.before:ehp2. starting the motor.
```

Output

Read **all** instructions **before** starting the motor.

:i1. / :i2. (Index)

Purpose

The primary and secondary index tags are used to place topics in the index.

Syntax

Tag	Element	Attributes	End
:i1.	Primary entry	id roots	
:i2.	Secondary entry	refid	

Attributes

id

Provides a cross-reference identifier for the i2 tag. This attribute is optional and is only valid when used with the primary index tag.

roots='root-words'

The roots attribute specifies a list of root-words which act as index entires to specified topics. These root-words are associated with the words you define with the index synonym tag. When defining the root-words, each word may contain alphabetic and numeric characters, which can be entered in upper or lowercase. When entering a string of words, a blank space is required between each word and the string of words must be enclosed in apostrophes.

The root-words do not appear in the index; therefore they are not viewed by the user. Root words are used to create a link between the primary index tag and the index synonym tag. To allow the user to search for an index entry, use the index synonym tag to map the root-words that are associated with the entry to synonyms.

Note: The word lists created by the root-words are used to link the index primary index tag to the index synonym tag and are not intended to be translated.

refid=

Provides a reference to the text associated with the primary index tag.

Description

The index is displayed when the user selects the index-entry text from a Help index list-panel or when the user selects Index from the search dialog box. If the user enters a synonym that matches a root-word, the index topics you listed for the root-word displays in the index.

You can provide index entries to the information by using the primary and secondary index tags, respectively. The attributes associated with each index entry allow you to define related information. Index entries can be used throughout the file, but they cannot be placed within a footnote.

The text of the index entry must be on the same line as the tag and cannot contain any other tags. The entry for each primary index entry within the panel must be unique. That is, you cannot provide duplicate index entries within the same panel. Secondary index entries must reference an identifier specified for a primary index entry.

Conditions

- Index entries cannot appear in a footnote.

Example

Example 1. This example shows how to tag your file to include primary and secondary index tags.

```
:i1 id=del.delete
:i2 refid=del.files
:i2 refid=del.directories
```

Output

delete
- **directories**
- **files**

Example 2. This example shows how to tag a file to include the *roots=' '* attribute.

```
:h1 id=copy03.Help for Copying
:isyn root=copy.copy copying duplicate duplicating
:isyn root=book.book manual draft manuscript
:isyn root=folder.folder folders document documents
:i1 roots='copy folder'.Copying a document
:i1 roots='book folder'.Test procedures
:p.When copying a file from the current directory to a new
directory, specify the following:
:ul.
:li.The file name
:li.The target directory
:li.The new file name and extension.
:eul.
```

Output: The index synonym tag creates a synonym table containing the following entries:

Root word	Synonym words
copy	copy copying duplicate duplicating
book	book manual draft manuscript
folder	folder folders document documents

The *roots=' '* points to the root-words *copy* and *folder* and the list of associated synonyms. For example, if the user searches for *copy* or *folder*, the *Copying a document* entry displays because *copy* and *folder*, identified by the index *roots=' '* attribute, matches the entries listed for the index synonym *root=' '* attribute.

A search for the synonym *duplicate* lists *Copying a document* as one of the index choices. A search for the synonym *manual* lists *Test procedures* as an index choice, and a search for *document* lists *Copying a document* and *Test procedures* as index choices.

:icmd. (Index command)

Purpose

The index command tag identifies the help panel that describes a command.

Syntax

Tag	Element	Attributes	End
:icmd.	Index command		

Attributes

external command string
The *external-command-string* specifies the command for which help is being defined. The text can contain no other tags.

Description

The help information for a command is assumed to be in the help panel in which the index command tag is defined. If the help panel provides help for more than a single command, an index command tag should be defined within the heading tag for each command. At run time, the index entries allow the compiler to process command helps, develop a list of commands for which help is available, and display the help panel defined for any of those commands.

The same *external-command-string* cannot be specified on more than one index command tag in an index file; that is, only one help panel can be designated as describing a command. If the same *external-command-string* is found more than once by the compiler (either from the same or different help panels), the duplicate occurrences are discarded and a warning message is issued.

Conditions

- This tag must follow a heading tag or another index tag.

Example

```
:h1 id=xhlp.Help for Copying
:icmd.Copying
⋮
:h1 res=129.Deleting Files
:icmd.Delete
```

Output

At run-time, the index entries allow the compiler to process command helps, create a list of commands for which help is available, and display the help panel defined for any of those commands.

.im (Imbed)

Purpose
Use this control word to specify the name of separate text or artwork files to be included at process time.

Syntax

Tag	Element	Attributes	End
.im	Imbed		

Attributes
None

Description
The imbed control word allows you to include text or artwork files when you are ready to compile your file.

Conditions
- If the imbed file is not in the current directory, you must enter a fully qualified file name.

Example
```
:userdoc.
.im filename.ext
.im c:\text\filename.ext
:euserdoc.
```

Output
The text and art contained in the imbedded files display when you access the compiled file.

:isyn. (Index synonym)

Purpose

The index synonym tag identifies word variations and synonyms for the help keywords.

Syntax:

Tag	Element	Attributes	End
:isyn.	Index synonyms	root	

Attributes

root = root-word
This attribute serves as a link between the various *synonym-words* and their corresponding index topic entry you create with the *roots=' '* attribute. When you enter the *root-word*, also enter the word in the list of synonym-words. The words entered in the synonym word-list allows the user to search for terms that may not be in the Index list, and still receive the appropriate help. To establish a link, specify the same *root-word* value as you specify in the *roots=* attribute of the primary index tag. A root-word can contain only alphabetic and numeric characters. Lower and uppercase characters are treated the same.

synonym-words
The *synonym words* specify variations and synonyms for the *root-word*. Use blanks to separate the synonym words.

Description

The index synonym tag begins an index entry for index help by specifying the data that will be used by the compiler.

Synonyms, which are linked to a primary index tag by the root-word determine the topic entries that display when the user enters words for a search of the Help index. The compiler uses the index synonym tag to build a table of synonyms. This table serves as a link to the primary index tags. When the user enters words for an index search, the compiler matches them with the words in this table to link to the topics to be displayed. Synonyms that you define with this tag do not appear in the Help index.

The index synonym tag can be placed within any panel that contains related index entries identified with the index tag. The synonyms defined in a help panel can relate to many topics, and thus to many help panels. The index synonym tag does not require an end tag.

Conditions

None

Example

```
:h1 id=copy03.Help for Copying
:isyn root=copy.copy copying duplicate duplicating
:isyn root=folder.folder folders document documents
:i1 roots='copy folder'.Copying a document
:p.When copying a file from the current directory to a new
directory, specify the following:
:ul.
:li.The file name
:li.The target directory
:li.The new file name and extension
:eul.
```

Output

The index synonym tag creates a synonym table containing the following entries:

Root word	**Synonym words**
copy	copy copying duplicate duplicating
folder	folder folders document documents

The attribute points to the root-words *copy* and *folder* and the list of associated synonyms. If the user searches for *copy* or *folder*, the entry displays because *copy* and *folder*, identified by the index *roots*=' ' attribute matches the entries listed for the index synonym *root*=' '. A search for the synonym *duplicate* gives *Copying a document* as an index choice.

:li. (List item)

Purpose
The list item tag identifies an item within an ordered list, an unordered list, or a simple list.

Syntax

Tag	Element	Attributes	End
:li.	List item		

Attributes
None

Description
The format of the list items depends on the type of list defined. For example, if the list item tag accompanies an unordered list, a bullet precedes each list item. Or if it accompanies an ordered list, a number precedes each list item. Each list item ends the previous list item, therefore an end tag is not required.

Conditions
None

Examples
```
:p.To remove a diskette:
:ol.
:li.Open the drive door.
:li.Remove the diskette.
:li.Put the diskette in a safe place.
:eol.
```

Output
To remove a diskette:

1. Open the drive door.
2. Remove the diskette.
3. Put the diskette in a safe place.

:lines. (Lines)

Purpose
The lines tags turn formatting off.

Syntax

Tag	Element	Attributes	End
:lines.	Lines	align	:elines.

Attributes

Align = left | right | center
This attribute places the lines on the left of the panel, on the right of the panel, or in center of the panel.

Description
The lines tags specify that text entered between the line and end line tags is formatted exactly as it is entered. Text that is too long to fit on the panel is clipped. Text is formatted using proportional fonts; therefore the text may not display exactly as entered. The attributes allow you to align text to the left, right, or center of the panel.

Conditions
None

Example
Example 1. This example aligns text to the left.

```
:lines align=left.
The warehouse contained:
  12  desks
  28  chairs
  15  lamps
  39  typewriters
  11  pictures
:elines.
```

Example 2. This example aligns text to the right.

```
:lines align=right.
The warehouse contained:
  12  desks
  28  chairs
  15  lamps
  39  typewriters
  11  pictures
:elines.
```

Output

Example 1.
The warehouse contained:
12 desks
28 chairs
15 lamps
39 typewriters
11 pictures

Example 2.

The warehouse contained:
12 desks
28 chairs
15 lamps
39 typewriters
11 pictures

:link. (Link)

Purpose

The link tag is used to specify hypertext or hypergraphic links.

Syntax

Tag	Link Elements	Attributes	End
:link.	Hypertext linking	reftype res refid object data	:elink.

Attributes

The reftype attributes allow links to headings, footnotes, and other text. The reftype attributes can also be used to start other programs or to send messages to an application. The *reftype=* attribute is always required with the link tag.

reftype=hd|fn|launch|inform

The reftype attribute identifies the elements you are linking to. The element specifies a link to a heading, a footnote, another program, or a message to the application.

(reftype=hd)	Headings
(reftype=fn)	Footnotes
(reftype=launch)	Start another program
(reftype=inform)	Send the application a message

res=

The resource identifier (res) attribute points to the integer defined by the heading or inform element. The resource identifier named with the link tag must match the resource identifier defined for the tag to which it is linking. When used with the inform element, the resource identifier identifies the integer sent to the application with the HM_INFORM message.

refid=

The reference identifier points to the ID specified by a tag. The reference identifier is required when linking from a footnote.

object=' '

The object attribute specifies the program you want to start and requires a fully qualified file name, including the directory, the name of the file, and the extension.

data=' '

The data attribute specifies the parameter strings for the program you are starting. Use of this attribute is dependent on the parameters specified for the program.

Description

The link tag allows you to link to heads and footnotes. You can also use the link tag to send a message to the application and to start another program. The link tag provides links to panels within the same database, but does not provide links to panels in separate databases.

The *reftype=' '* attribute is required with each link tag description. This attribute identifies the type of link you are defining. You must also include any other attribute specified for the element from which you are linking.

The *object=' '* attribute is required when the element is launch. The information you enter for this attribute must include a fully qualified file name, including the extension of the program you want to start. If the program requires data parameters, they must also be included, using the *data=' '* attribute.

The inform element allows messages to be sent to the application. The *res=' '* attribute identifies the message. The compiler sends an HM_INFORM message with the *res=' '* identifier to the application. The action taken by the application is predefined in the application's code.

The link attributes are used to link both text and artwork. When linking from artwork, include the values for the hypergraphic area if the values are required.

When specifying hypergraphic links, use the link tag to specify a link from a piece of art just as you would when specifying hypertext links. The difference is where you place the information and the link format you use.

When creating hypertext links, enter:

- The link tag
- The reference type
- The identifier of the element you are linking from
- The text to be highlighted
- The end link tag.

Notice that the link tags start and end the link format.

When creating hypergraphic links,

- Define the link definition file, using the *linkfile=' '* attribute of the artwork tag.
- Begin the link definition file by entering the artlink tag. Then enter:
 - The link tag
 - The reference type
 - The identifier of the element you are linking to
 - The object and data parameters, if required
 - The hypergraphic values, if required
 - The end artlink tag.

Notice that the link information is placed between the artlink and end artlink tag. The end link tag is omitted.

Conditions

- The link tag requires an end link tag when creating hypertext, but not hypergraphics.
- Launch requires the *object=' '* attribute and a fully qualified file name.
- Inform requires the *res=* attribute.

Example

Examples 1, 2, and 3 show how to tag when linking from text.

Example 1. To link from a head, you could enter:

```
Click on
:link. res=150 reftype=hd. Copy format :elink.
```

Example 2. To send a message to the application, include the following tags:

```
Click on
:link res=33 reftype=inform.
Send a message.
:elink.
```

Example 3. To start another program, include the following tags:

```
Click on
:link reftype=launch object='c:\apps\demo.exe' data='test4'.
Run demo.exe
:elink.
```

Examples 4 and 5 show how to tag when linking from art.

Example 4. To link from a whole bit map, you could tag the links as follows:

```
:artlink.
:link reftype=hd refid=fmt321.
:eartlink.
```

Example 5. To link from a segmented bit map, you could tag the links as follows:

```
:artlink.
:link reftype=hd res=001 x=0 y=0 cx=16 cy=8.
:link reftype=fn refid=afnr x=16 y=8 cx=16 cy=8.
:link reftype=inform res=0345 x=0 y=8 cx=16 cy=8.
:link reftype=launch object='c:\os2\e.exe' data='c:\appsdir\tutor.dat'
        x=16 y=0 cx=16 cy=8.
:eartlink.
```

Output

Example 1. In this example, the words *Copy format* are highlighted; clicking on them displays the page with the heading associated with the res = 150.

Example 2. In this example, when the user clicks on *Send a message*, an HM_INFORM message with the integer *res = 33* is sent to the application.

Example 3. In this example, the link is a request to start the program **demo.exe** and pass the parameter *test4*.

Example 4. In this example, the link is from a whole bit map. Because the link if from a whole bit map, the bit map area or the hypergraphic values specifying the location of the bit map are not required and are omitted.

Example 5. This example shows how to provide links from a segmented bit map. The hypergraphic value specifying the location of each bit map segment is included in the link definitions.

:lm. (Left margin)

Purpose
The left margin tag sets the left margin of the text.

Syntax

Tag	Element	Attributes	End
:lm.	Left margin	margin	

Attributes

margin=

The *margin=* attribute allows you to change the beginning margin of the text. Use any number with this attribute to specify where the left margin of the text begins. To set the margin for the current line, specify a number that is greater than the position of the cursor. For example, to set the left margin to 15, begin the left margin tag before space 15. Left margins specified after the current cursor position are effective on the next line.

Note: In this document character spaces or spaces mean the same as *average character widths.* See "Graphic Text" in this book for details and examples.

Description
Use the left margin tag and the right margin tag to specify the boundaries of the text on the panel. Setting margins on the panel works the same as setting margins on a typewriter. The left margin shows where you want the text to start, and the right margin shows where you want the text to end.

To set a left margin for the current line, begin the left margin tag before the boundary you want to specify for the margin. For example, if you wanted to set the left margin to 12, begin the left margin tag before space 12 on the current line. When the text following the left margin tag formats, the text begins on the line at space 12. Other text aligns with the margin specified. If you start the left margin tag after the boundary indicated by the margin value, the margin becomes effective on the following line.

When the help window is sized, the text area adjusts from the right to fit within the margin boundaries specified; that is, the right margin reformats to the new panel size. The left margin remains constant. If the panel is sized smaller than the margins specified, the margins remain the same and the text area is reduced to one character space.

You can place multiple margin tags in your file. The margins specified remain effective until the margin is reset. If no margin value is specified, the default is one. The left margin tag does not require an end tag.

Conditions
None

Example

```
:p.
:rm margin=10.
:lm margin=20.This text begins 20 spaces to the right of the left
panel border and ends 10 spaces to the left of the right panel border.
All text is aligned as specified
by the margin values. :lm=margin=5. Here the left margin is
changed to 5.  Because this margin tag begins
more than 5 spaces on the line, the margins specified become effective
on the following line and the text begins 5 spaces from the left panel
border.  The right margins remain unchanged.
```

Output

This text begins 20 spaces to right of the left panel border and ends 10 spaces to the left of the right panel border. All text is aligned as specified by the margin values. Here the left margin is changed to 5. Because this margin tag begins more than 5 spaces on the line, the margins specified become effective on the following line and the text begins 5 spaces from the left panel border. The right margins remain unchanged.

:lp. (List part)

Purpose
The list part tag identifies an explanation within an ordered list, an unordered list, or a simple list.

Syntax

Tag	Element	Attributes	End
:lp.	List part		

Attributes
None

Description
The list part tag can be entered anywhere within the list. The text following the list part tag starts at the left margin of the current level of the list. It is not numbered or lettered. Using the list part tag does not interrupt the sequence of the list. The list part tag requires no end tag.

Conditions
None

Examples
To remove a diskette

```
:ol.
:li.Open the drive door.
:lp.Before removing the diskette, make sure all drive activity
has ceased.
:li.Remove the diskette.
:li.Put the diskette in a safe place.
:eol.
```

Output
To remove a diskette

1. Open drive door.

 Before removing the diskette, make sure all drive activity has ceased.
2. Remove the diskette.
3. Put the diskette in a safe place.

:note. (Note)

Purpose
The note tag defines a note.

Syntax

Tag	Element	Attributes	End
:note.	Note	text	

Attributes
text=' '
This attribute allows you to change the name of the note.

Description
The note tag identifies a single paragraph note and like the paragraph tag, it causes a blank line before the tagged area. The start of another tag ends the note, therefore it requires no end tag.

The text of the note formats as an implied paragraph, beginning at the left margin. The word "Note:" begins the paragraph and is followed by two blank spaces. When used within a list, the note aligns with the text of the items within the list. Use the *text=' '* attribute to give the note a specific name.

Conditions
None

Example
```
:note.
This text appears within a note.
The word Note aligns
with the text that precedes it.
```

The following example uses the *text=' '* attribute.

```
:note text='Text note:'.
The name of this note is Note Text.
The text for the note replaces
the word Note.  The name of the note
aligns with the text that precedes it.
```

Output
Note: This text appears within a note. The word *Note* aligns with the text that precedes it.

The following example uses the *text=' '* attribute.

Text note: The name of this note is Note Text. The text for the note replaces the word *Note*. The name of the note aligns with the text that precedes it.

:nt. (Note)

Purpose
The :nt. tag defines a note that can have multiple paragraphs.

Syntax

Tag	Element	Attributes	End
:nt.	Note	text	:ent.

Attributes
text=' '
This attribute allows you to change the name of the note.

Description
The text of a note begun with the :nt tag formats as an implied paragraph, beginning at the left margin. The word "Note:" begins the paragraph and is followed by two blank spaces. Notes can be placed within lists and paragraphs. However, unlike the :note. tag, the :nt. tag requires an end tag. When the file containing the :nt. tag is displayed, the word *Note* aligns with the text preceding it and the text of the note begins to the right of the word *Note*. You can use the *text=' '* attribute to assign a specific name to the note. This tag causes a blank line before the tagged text.

Conditions
None

Example
```
:nt.
Use this tag to include paragraphs in a note.
You can also use it within
paragraphs and lists.

End this tag before you begin another note tag.
:ent.
```

Output
Note: Use this tag to include paragraphs in a note. You can also use it within paragraphs and lists.

End this tag before you begin another note tag.

:ol. (Ordered list)

Purpose
The ordered list tag defines a sequential set of items or steps.

Syntax

Tag	Element	Attributes	End
:ol.	Ordered list	compact	:eol.

Attributes

compact
The compact attribute causes lists to format without blank lines between the lines of text. If this attribute is omitted, a blank line appears between each list item.

Description
Use the list item tag to place items in the list. Use the list part tag to include a paragraph in the list. Ordered lists format as indented lists, with the sequential numbers or letters at the left of each list item. Ordered lists can be nested or imbedded within other lists. When an ordered list is nested within another ordered list, the first list has sequential numbers at the left margin and the second list has sequential letters indented two spaces. After the second list, the number-letter sequence is repeated for each successive ordered list. When nesting lists, be sure to end each list with the appropriate end list tag.

Conditions
None

Example

```
:p.To remove a diskette:
:ol.
:li.Open the drive door.
:li.Remove the diskette.
:li.Put the diskette in a safe place.
:eol.
```

Output
To remove a diskette:

1. Open the drive door.
2. Remove the diskette.
3. Put the diskette in a safe place.

:p. (Paragraph)

Purpose
The paragraph tag begins a new paragraph.

Syntax

Tag	Element	Attributes	End
:p.	Paragraph		

Attributes
None

Description
Each paragraph identified by a paragraph tag formats as an unindented block of text. The compiler adds a blank line before the paragraph. Paragraphs placed within a list align with the text of the list. When paragraphs are placed within a note, the text of the paragraph aligns with the text of the note. The paragraph tag requires no end tag.

Conditions
None

Example

```
:p.Paragraph tags cause a blank line before the text.
When placed within a list or note, the text of the paragraph
aligns with the text of the note.
:ul.
:li.Paragraph tags
:p.Paragraph tags are flexible and can be used
with most tags.
:li.Note tags
:p.Note tags can include paragraphs.
:eul.
```

Output

Paragraph tags cause a blank line before the text. When placed within a list or note, the text of the paragraph aligns with the text of the list or note.

- Paragraph tags

 Paragraph tags are flexible and can be used with most tags.

- Note tags

 Note tags can include paragraphs.

:parml. (Parameter list)

Purpose

The parameter list tag defines a two-column list of terms and descriptions.

Syntax

Tag	Element	Attributes	End
:parml.	Parameter list	tsize break compact	:eparml.

Attributes

tsize = 10 | n

The *tsize* = attribute specifies the space allocated for the parameter term. The default is 10 character units.

break = all | fit | none

The *break* = attribute controls the formatting of the parameter terms and descriptions. This attribute uses the following values:

break = all

When this value is specified, the description begins on the line below the parameter term in the space allocated by the tsize attribute.

break = fit

When this value is specified, the parameter description begins on the same line as the parameter term if the parameter term contains fewer characters than specified in the tsize attribute. If the parameter term contains more characters than specified in the tsize, the description formats on the line below the term.

break = none

When this value is specified, the parameter description begins on the same line as the parameter term. If the parameter term contains more characters than specified in the tsize, the term continues on into the parameter description area. The parameter description starts one space after the parameter term ends.

compact

The *compact* attribute instructs the compiler to format the list without a blank line between the items.

Description

Parameter lists are similar to definition lists, defining terms and descriptions that format in two-columns. The elements of the parameter list tag are the parameter list tag and its matching end tag, the parameter term tag, and the parameter description tag. As in definition lists, the term tag identifies a term, and the definition tag identifies the description associated with the term. Each parameter term tag requires an accompanying parameter description tag. When tagging the file, the parameter description tag follows the parameter term tag.

Parameter lists can occur anywhere in text; you can nest them within other lists, and you can nest other lists within parameter lists. When nesting lists, each list must have a beginning and an ending list tag.

Conditions

- Each parameter term tag requires an accompanying parameter description.

Example

```
:parml compact tsize=10.
:pt.Tree
:pd.Plant life in forest
:pt.Orange
:pd.Fruit on tree
:pt.Cow
:pd.Animal on farm
⋮
:eparml.
```

Output

Tree
: Plant life in forest

Orange
: Fruit on tree

Cow
: Animal on farm

:pd. (Parameter description)

Purpose
The parameter description tag identifies the description for each term in a parameter list.

Syntax

Tag	Element	Attributes	End
:pd.	Parameter description		

Attributes
None

Description
The text that follows the parameter description tag describes the term that is identified by the parameter term tag. Like the definition description, the parameter description formats in the right column. The column format depends on the values specified in the *tsize=* and the *break=* attributes. See the "Parameter list" tag for more information.

When tagging the file, the parameter description follows the parameter term. A parameter list can contain multiple parameter term and parameter description tags. However, each parameter term tag requires an accompanying parameter description tag. The parameter description tag is valid only within parameter lists. The parameter description tag requires no end tag.

Conditions
- The parameter description tag is paired with a parameter term tag.
- The parameter description tag follows the parameter term tag.
- The parameter description tag is valid only within a parameter list.

Example
```
:parml compact tsize=10.
:pt.Tree
:pd.Plant life in forest
:pt.Orange
:pd.Fruit on tree
:pt.Cow
:pd.Animal on farm
:
:eparml.
```

Output
Tree
 Plant life in forest
Orange
 Fruit on tree
Cow
 Animal on farm

:pt. (Parameter term)

Purpose
The parameter term tag identifies the term in a parameter list.

Syntax

Tag	Element	Attributes	End
:pt.	Parameter description		

Attributes
None

Description
Like the definition term tag, the term identified by the parameter term tag formats in the left column. The information describing the term formats in the right column. This information is identified by the parameter description tag. Each term in a parameter list begins with the parameter term tag; and each parameter term tag is paired with a parameter description tag. When tagging the file, the parameter term tag precedes the parameter description tag. Parameter tags are valid only within parameter lists. Parameter lists can contain multiple parameter term and parameter descriptor tags. See the "Parameter list" tag for more information.

Conditions
- The parameter term tag is paired with a parameter description tag.
- The parameter term tag precedes the parameter description tag.
- The parameter term tag is valid only within a parameter list.

Example
```
:parml compact tsize=10.
:pt.Tree
:pd.Plant life in forest
:pt.Orange
:pd.Fruit on tree
:pt.Cow
:pd.Animal on farm
:
:eparml.
```

Output
Tree
 Plant life in forest
Orange
 Fruit on tree
Cow
 Animal on farm

:rm. (Right margin)

Purpose
The right margin tag sets the right margin of the text.

Syntax

Tag	Element	Attributes	End
:rm.	Right margin	margin	

Attributes

margin=

The *margin=* attribute allows you to indicate where the text ends. Use any number with this attribute to specify where the text should end. When specifying the margins the left margin count begins from the left of the panel; however, the right margin count begins from the right side of the panel. So, to set the right margin to 60 means that you want the text to end 60 spaces from the right of the panel.

Note: When counting character spaces you are actually counting *average character widths.* For details and examples about average character widths, see "Graphics Text" in this book.

Description
Use the right margin tag and the left margin tag to specify the boundaries of the text on the panel. Setting margins on the panel works the same as setting margins on a typewriter. The left margin shows where you want the text to start and the right margin shows where you want the text to end. When setting right margins, remember that the count is from high to low. It is just the opposite of setting left margins, where the count is from low to high.

Margin tags can be set at the beginning of the line of text or while entering the text. Margin tags that begin the line of text cause text on that line and the following lines to align with the margin values specified. Margins set while entering text can become effective on the current line or on the next line, depending on where the margin tag begins. If the margin tag and the value are entered before the boundary specified, the margin is effective on the current line. For example, to set the right margin to 60, (that is, 60 spaces before the right panel border) begin the right margin tag at least 60 spaces to the left of the right panel border. When the file is displayed, the text entered after the margin tag is aligned to the value specified on that line.

If the margin tag is entered after the boundary specified, the margin is effective on the next line. For example, if the cursor is positioned 60 spaces before the right panel border and the right margin is set to 65 (that is 65 spaces before the right panel border), the margin becomes effective on the next line because from the position of the cursor, there are only 60 spaces remaining before reaching the right panel border.

When the help window is sized, the text area adjusts from the right to fit within the margin boundaries specified; that is, the right margin reformats to the panel size. The left margin remains constant. If the panel is sized smaller than the margins specified, the margins remain the same and the text area is reduced to one character space. If no margin value is specified, the default for the right margin is one.

You can place multiple margin tags in your file. The margins specified remain effective until the margins are reset. The right margin tag does not require an end tag.

Conditions
None

Example

```
:lm margin=1.
:rm margin=44.
:p.In this
example the left margin is 1. The right margin
is 44.  The margins are set before the text; therefore
when the file is displayed, the text formats according
to the margins set.  The text begins at space 2 and
ends 44 spaces before the right panel border.  If the
margin specified is less than the current cursor position
on the screen, the margins set become effective on the following
line.  For example, if the current cursor position is 60 spaces to
the left of the right panel border and you set the right margin to
50, the margin is effective on the current line.  If, however, the
margin is set to 65, the margins become effective on the next line.
:p.
:lm margin=5.
:rm margin=60. Here the left margin is set to 5 and the right
margin is set to 60.  That means the left margin begins 5 spaces to
the right of the left panel border.  The right margin ends 60 spaces
to the left of the right panel border.
```

Output

In this example the left margin is 1. The right margin is 44. The margins are set before the text; therefore when the file is compiled, the text formats according to the margins set. The text begins at space 2 and ends 44 spaces before the right panel border. If the margin specified is less than the current cursor position on the screen, the margins set become effective on the following line. For example, if the current cursor position is 60 spaces to the left of the right panel border and you set the right margin to 50, the margin is effective on the current line. If, however, the margin is set to 65, the margins become effective on the next line.

Here the left margin is set to 5 and the right margin is set to 60. That means the left margin begins 5 spaces to the right of the left panel border. The right margin ends 60 spaces to the left of the right panel border.

:sl. (Simple list)

Purpose
This tag defines a nonsequential list of items.

Syntax

Tag	Element	Attributes	End
:sl.	Simple list	compact	:esl.

Attributes

Compact
The *compact* attribute causes the list to format without blank lines between the list items.

Description
The simple list tag identifies items that do not require a sequential listing. When compiled, items in a simple list are indented without bullets, hyphens, or dashes preceding them. Simple lists can be nested within other lists. When nested, simple lists indent four spaces to the right of the left margin of the lists that contain them.

The simple list tag uses the list item tag to identify items and the list part tag to include paragraphs in a simple list. The simple list ends with the end simple list tag.

Conditions
None

Example

```
Bring the following for lunch:
:sl compact.
:li.Fruit, for example:
:sl compact.
:li.An apple
:li.An orange
:li.A pear
:li.A banana
:esl.
:li.Sandwich
:li.A drink, for example:
:sl compact.
:li.A soda
:li.Juice
:li.Milk.
:esl.
:esl.
```

Output

Bring the following for lunch:

Fruit, for example:
- An apple
- An orange
- A pear
- A banana

Sandwich

A drink, for example:
- A soda
- Juice
- Milk.

:ul. (Unordered list)

Purpose
The unordered list tag identifies a list of nonsequential items.

Syntax

Tag	Element	Attributes	End
:ul.	Unordered list	Compact	:eul.

Attributes

Compact
The *compact* attribute causes the list to format without blank lines between the list items.

Description
The unordered list tag identifies items that do not require a sequential listing. The list item tag identifies the items within the unordered list and the list part tag is used to include paragraphs within the list. When compiled, unordered list items are indented and a bullet precedes each item. Unordered lists can be placed within other lists. When placed within another unordered list, the list items in the second list are indented four spaces to the right. That is, items in the second list begin four spaces to the right of the left margin of the list that contains them. The items in the first list are preceded by a bullet (•). Items in the second list are preceded by a hyphen (-). If more lists are included, the bullet / hyphen sequence is repeated with each successive list.

Conditions
None

Example

```
Before leaving for the day remember to:
:ul.
:li.Turn off the computer
:li.Turn off the lights
:lp.Turning off the light and equipment:
:ul.
:li.Reduces electrical waste
:li.Reduces electrical costs
:eul.
:li.Secure all equipment.
:eul.
```

Output

Before leaving for the day remember to:

- Turn off the computer
- Turn off the lights

 Turning off the light and equipment:

 - Reduces electrical waste
 - Reduces electrical costs
- Secure all equipment.

:userdoc. (User document)

Purpose

The userdoc tags identify the markup file that is to be compiled.

Syntax

Tag	Element	Attributes	End
:userdoc.	User Document		:euserdoc.

Attributes

None

Description

The userdoc tag must be the first tag in the file. It signals the compiler to begin compiling the tagged text that follows. All other tags that define how the text is to format follow this tag. The end user document tag identifies the end of the tagged text and the end of the markup file. It must be the last tag in the file.

Conditions

None

Example

```
:userdoc.
⋮
:euserdoc.
```

:warning. (Warning)

Purpose

Use the warning tags to alert the user of a risk or possible error condition.

Syntax

Tag	Element	Attributes	End
:warning.	Warning	text	:ewarning.

Attributes

text=' '
This attribute allows you to give a name to the warning.

Description

The warning, like the caution, alerts the user to a possible risk, such as an error condition in the system. The warning statement should appear before the text that it discusses. Use the *text=' '* attribute to provide a specific warning notice.

Conditions

None

Example

```
:warning.
The disk contains bad sectors.
:ewarning.
```

To provide specific warning text, you could enter:

```
:warning text='Bad disk:'.
The disk contains bad sectors.
:ewarning.
```

Output

Warning: The disk contains bad sectors.

The specific warning text looks like this:

Bad disk: The disk contains bad sectors.

:xmp. (Example)

Purpose
The example tags turn formatting off.

Syntax

Tag	Element	Attributes	End
:xmp.	Example		:exmp.

Attributes
None

Description
Text entered between the example and end example tags formats as it is entered in monospace fonts. Lines that are too long to fit within the panel are clipped. Example text is indented two spaces from the left margin of the panel. The example tag has no attributes and requires an end example tag.

Conditions
- An example cannot be placed within another example.

Example
```
:xmp.
:p.The text shown here is in example format.
:p.This example contains two paragraph
tags.  Like the lines tag, the text formats
as it is entered for the example.

Complete the following.
The answers are below.

                Pattern
               (2, 4, 9)
              (7, 14, 21)

    Using the pattern, fill in the missing numbers

         a.          b.          c.
   1.   (  )        ( 30)       ( 45)
   2.   (50)        (   )       (175)
   3.   (  )        (120)       (   )
   4.   (90)        (   )       (   )


:p.The answers are:
1a.15; 2b.100; 3a.60; 3c.210; 4b.180; 4c.315.
:exmp.
```

Output

```
The text shown here is in example format.

This example contains two paragraph
tags.  Like the lines tag, the text formats
as it is entered for the example.
Complete the following.
The answers are below.

             Pattern
            (2, 4, 9)
           (7, 14, 21)

   Using the pattern, fill in the missing numbers

        a.           b.           c.
   1.   (  )        ( 30)        ( 45)
   2.   (50)        (   )        (175)
   3.   (  )        (120)        (   )
   4.   (90)        (   )        (   )


The answers are:
1a.15; 2b.100; 3a.60; 3c.210; 4b.180; 4c.315.
```

Appendix I. Information Presentation Facility Symbols

This appendix discusses the symbols you can use to display special characters that you may want to include in your file. Symbols can be used to specify characters that are not on the keyboard. Each symbol represents a single character. When tagging your file to include symbols, begin each symbol with an ampersand (&) and end the symbol with a period (.). For example, to place a square bullet (■) in a file, you would enter:

```
&sqbul.
```

Symbols are case sensitive, that is uppercase characters produce different symbols than lowercase characters. Therefore, when tagging the file to include a symbol, enter the tag for the symbol exactly as it is shown in the symbols chart.

Note: All symbols in the following chart are also in the `APSYMBOL.APS` file. This file is in the C:/TOOLKIT12/IPFC directory and can be edited with any text editor.

Symbol	Element	Character
&aa.	a acute	á
&Aa.	A acute	Á
&ac.	a circumflex	â
&ae.	a umlaut	ä
&Ae.	A umlaut	Ä
&ag.	a grave	à
&aelig.	ae ligature	æ
&AElig.	AE ligature	Æ
&alpha.	alpha	α
&Alpha.	Alpha	A
&amp.	Ampersand	&
&and.	and	∧
&angstrom.	angstrom	Å
&ao.	a overcircle	å
&Ao.	A overcircle	Å
&apos.	Apostrophe	'
&bx1012.	ASCII code 181	╡
&bx2021.	ASCII code 182	╢
&bx0021.	ASCII code 183	╖
&bx0012.	ASCII code 184	╕
&bx2022.	ASCII code 185	╣
&bx2020.	ASCII code 186	║
&bx0022.	ASCII code 187	╗
&bx2002.	ASCII code 188	╝
&bx2001.	ASCII code 189	╜
&bx1002.	ASCII code 190	╛
&bx1210.	ASCII code 198	╞
&bx2120.	ASCII code 199	╟
&bx2200.	ASCII code 200	╚
&bx0220.	ASCII code 201	╔
&bx2202.	ASCII code 202	╩

Symbol	Element	Character
&bx0222.	ASCII code 203	╦
&bx2220.	ASCII code 204	╠
&bx0202.	ASCII code 205	═
&bx2222.	ASCII code 206	╬
&bx1202.	ASCII code 207	╧
&bx2101.	ASCII code 208	╨
&bx0212.	ASCII code 209	╤
&bx0121.	ASCII code 210	╥
&bx2100.	ASCII code 211	╙
&bx1200.	ASCII code 212	╘
&bx0210.	ASCII code 213	╒
&bx0120.	ASCII code 214	╓
&bx2121.	ASCII code 215	╫
&bx1212.	ASCII code 216	╪
&asterisk.	Asterisk	*
&atsign.	At sign	@
&bslash., &bsl.	Back slash	\
&Beta.	Beta	B
&bxas., &bxbj.	box ascender	┴
&bxcr., &bxcj.	box cross	┼
&bxde., &bxtj.	box descender	┬
&bxh.	box horizontal	─
&bxll.	box lower-left	└
&bxlr.	box lower-right	┘
&bxri., &bxrj.	box right junction	┤
&bxul.	box upper-left	┌
&bxur.	box upper-right	┐
&bxv.	box vertical	│
&bullet.	bullet	•
&cc.	c cedilla	ç
&Cc.	C cedilla	Ç
&caret.	Caret symbol	^
&cent	cent	¢
&cdq.	Close double quote	”
&cdqf.	close french double quote	»
&csq.	Close single quote	’
&comma.	Comma	,
&colon.	Colon	:
&dash.	Dash	–
&degree., &deg.	degree	°
&delta.	delta	δ
&divide	divide	÷
&dollar.	Dollar sign	$
&dot.	dot	.
&darrow.	Down arrow	↓

Symbol	Element	Character
&ea.	e acute	é
&Ea	E acute	É
&ec.	e circumflex	ê
&ee.	e umlaut	ë
&eg.	e grave	è
&emdash.	Em dash	—
&endash.	En dash	–
&epsilon.	epsilon	ε
&eq., &equals., &eqsym.	Equal sign	=
&xclm., &xclam.	Exclamation point	!
&fnof.	function of	*f*
&Gamma.	Gamma	Γ
&gtsym., &gt. &gesym.	Greater than	>
&house.	House	⌂
&hyphen.	Hyphen	-
&ia.	i acute	í
&ic.	i circumflex	î
&identical.	identical	≡
&ie.	i umlaut	ï
&ig.	i grave	ì
&infinity.	infinity	∞
&inttop.	integral top half	⌠
&intbot.	integral bottom half	⌡
&intersect.	intersect	∩
&inve.	inverted exclamation mark	¡
&invq.	inverted question mark	¿
&larrow.	Left arrow	←
&lbrace., &lbrc.	Left brace	{
&lbracket. &lbrk.	Left bracket	[
&lpar. , &lparen.	Left parenthesis	(
&lt., &ltsym.	Less than	<
&le., &lesym.	less than or equal to	≤
&lnot.	logical not	¬
&lnotrev.	backwards logical not	⌐
&mdash.	M dash	—
&minus.	Minus sign	–
&mu.	Mu	*μ*
&ge., &gesym.	much greater than	≥
&ndash.	N dash	–
&nt.	n tidle	ñ
&Nt.	N tidle	Ñ
&nearly.	nearly	≈
&lnot., &notsym.	not symbol	¬
&numsign.	Number sign	#
&oa.	o acute	ó

Symbol	Element	Character
&oc.	o circumflex	ô
&og.	o grave	ò
&oe.	o umlaut	ö
&Oe.	O umlaut	Ö
&Omega.	Omega	Ω
&frac14.	one fourth	¼
&frac12.	one half	½
&odq.	Open double quote	“
&odqf.	open french double quote	«
&osq.	Open single quote	‘
&percent.	Percent	%
&per.	Period	.
&phi.	phi	ϕ
&Phi.	Phi	Φ
&pi	Pi	π
&plus.	Plus sign	+
&plusmin., &pm.	plusminus	±
&Lsterling.	pound sterling	£
&rahead	right arrowhead	▸
&rarrow	right arrow	→
&rbl.	Required blank	
&rbrace., &rbrc.	Right brace	}
&rbracket., &rbrk.	Right bracket	]
&rpar., &rparen.	Right parenthesis	)
&semi.	Semicolon	;
&box14.	shaded box 1/4 dots	⣿
&box12.	shaded box 1/2 dots	▒
&box34.	shaded box 3/4 dots	▓
&sigma.	sigma	σ
&Sigma.	Sigma	Σ
&slash., &slr.	Slash	/
&BOX.	solid box	▉
&BOXBOT.	solid box bottom half	▄
&BOXLEFT.	solid box left half	▌
&BOXRIGHT.	solid box right half	▐
&BOXTOP.	solid box top	▀
&splitvbar.	Split vertical bar (piping symbol)	¦
&sqbul.	square bullet	■
&sqrt.	square root	√
&supn.	superscript n	^{n}
&sup2.	superscript 2	^{2}
&Theta.	Theta	Θ
&tilde.	Tilde	~
&tau.	Tau	τ
&ua.	u acute	ú

Symbol	Element	Character
&uc.	u circumflex	û
&ug.	u grave	ù
&uarrow	up arrow	↑
&ue.	u umlaut	ü
&Ue.	U umlaut	Ü
&us.	Underscore	_
&aus.	underscored a	a̲
&ous.	underscored o	o̲
&ye.	y umlaut	ÿ
&yen.	yen	¥

Appendix J. OS/2 1.2 Conformance with Common User Access

The following table lists by component the OS/2 1.2 deviations from the Common User Access, as documented in *Systems Application Architecture, Common User Access: Advanced Interface Design Guide*, SC26-4582, June, 1989.

Component	Deviation
SYSTEM INSTALL	Is text-based, not graphical-based
SYS LOG	Is text-based, not graphical-based
INSTALL AID	Is text-based, not graphical-based
TOOLKIT INSTALL	Is text-based, not graphical-based
TUTORIAL	1. Main panel does not have system menu or window-sizing icons 2. Has Function Key Area 3. Selection cursor is not correct 4. Colors do not conform
PIC UTILITIES	1. No object in initial client area 2. First item in File pull-down is not **Open...** 3. Uses brackets around directory names in the Directory list 4. **File name:** and **Directory is:** are in the wrong order in dialogs
SHELL	1. Control panel is not sizable 2. Control panel does not support object/action 3. Control panel does not support Maximize 4. Task Manager and Desktop Manager do not support Close 5. Scroll bars disappear when not required in File Manager 6. De-select method differs in the File Manager
SPOOLER	1. Enter is used as a pushbutton 2. Field prompts are not followed by colons 3. Message boxes do not have titles and icons 4. Destructive default action is used in message boxes 5. Forms dialog box does not conform
PM	1. List boxes do not support extended selection 2. Visual appearance differs from the published guide
HM/ONLINE REF	1. Contents and Index do not have horizontal scroll bars 2. No drag select 3. No *H to Hawaii* support 4. No Window action bar choice with Tile and Cascade options 5. De-select method differs
UTILITIES	Are in VIO windows and do not have an Action Bar for CHKDSK, COPY, LABEL, FORMAT, and COMPARE
FDISK	1. Capitalization on pull-downs is not correct 2. System menus for messages and dialog boxes are not correct 3. About... window does not have an FDISK icon 4. Tab is not supported in Close message box

Component	Deviation
DM	Does not support: 1. Secondary windows or modeless dialog boxes 2. Extended selection 3. Direct manipulation or drag operations 4. Clipboard 5. CUA-defined interactions on Save As dialog box 6. Truncating window titles 7. Automatic placement of key engravings in pull-down choices (application must hard code it) 8. Bit maps or icons in pull-downs 9. Modified pull-down choices 10. Cascading pull-downs 11. Ability to provide additional scrolling information adjacent to the horizontal scroll bar 12. Ability for the application to specify the scroll amount used in page scrolling 13. Cursor-driven scrolling 14. Filling check boxes with gray to indicate indeterminate state 15. Auto-Tab 16. Combination boxes 17. Drop-down combination boxes 18. Drop-down lists 19. Grayed entry fields 20. Field prompts for list boxes 21. Group boxes 22. Group headings 23. Support to change appearance of the mouse cursor 24. Progress indicator controls 25. Message icons 26. CUA-defined pushbuttons in message pop-ups 27. Timed-display of the logo panel 28. Multiple Document Interface 29. Graying of scroll bars when inactive (DM hides the scroll bars) Also, DM calls an action message an error message.

Glossary

This glossary defines terms used in this book. It contains terms specific to the Presentation Manager, but it is not a complete glossary for OS/2 Version 1.2. This glossary includes terms and definitions from the *IBM Dictionary of Computing*, SC20-1699.

A

accelerator. A single key stroke that invokes an application-defined function.

accelerator table. Used to define which keystrokes are treated as accelerators, and the commands they are translated into.

action. One of a set of defined tasks a computer performs. Users request the application to perform an action in several ways: typing a command, pressing a function key, or selecting the action name from an action bar or menu.

action bar. The area at the top of a panel that contains keywords that give users access to actions available in the current panel. When users select an action bar choice, a group of actions or additional keywords appear in a pull-down extension from the action bar.

action point. The current position on the screen at which the pointer is pointing. (Contrast with *hot spot* and *input focus*.)

active program. A program currently running on the computer. See also *interactive program*, *noninteractive program*, and *foreground program*.

active window. The window with which the user is currently interacting.

alphanumeric video output. Output to the logical video buffer when the video adapter is in text mode and the logical video buffer is addressed by an application as a rectangular array of character cells.

anchor block. An area of Presentation Manager-internal resources allocated to a process or thread that calls WinInitialize.

anchor point. A point in a window used by a program designer or by a window manager to position a subsequently appearing window.

ANSI. American National Standards Institute.

APA. All points addressable.

API. Application programming interface. The formally-defined programming language that is between an IBM application program and the user of the program. See also GPI.

area. In computer graphics, a filled shape such as a solid rectangle.

ASCII. American National Standard Code for Information Interchange.

ASCIIZ. A string of ASCII characters that is terminated with a byte containing the value 0.

aspect ratio. In computer graphics, the width-to-height ratio of an area, symbol, or shape.

asynchronous. (1) Without regular time relationship. (2) Unexpected or unpredictable with respect to the execution of a program's instructions.

atom. A constant that represents a string. Once a string has been defined as an atom, the atom can be used in place of the string to save space. Strings are associated with their respective atoms in an **atom table**. See also **integer atom**.

atom table. Used to relate **atoms** with the strings that they represent. Also in the table is the mechanism by which the presence of a string can be checked.

attributes. Characteristics or properties that can be controlled, usually to obtain a required appearance; for example, the color of a line. See also *graphics attributes* and *segment attributes*.

AVIO. Advanced Video Input/Output.

B

background color. The color in which the background of a graphic primitive is drawn.

background mix. An attribute that determines how the background of a graphic primitive is combined with the existing color of the graphics presentation space. Contrast with *mix*.

Bezier curves. A mathematical technique of specifying smooth continuous lines and surfaces, which require a starting point and a finishing point with several intermediate points that influence or control the path of the linking curve. Named after Dr. P. Bezier.

bit map. A representation in memory of the data displayed on an APA device, usually the screen.

border. A visual indication (for example, a separator line or a background color) of the boundaries of a window.

button. A mechanism on a *pointing device*, such as a mouse, used to request or initiate an action. Contrast with *pushbutton* and *radio button.*

C

cached micro presentation space. A presentation space from a Presentation Manager-owned store of micro presentation spaces. It can be used for drawing to a window only, and must be returned to the store when the task is complete.

cancel. An action that removes the current window or menu without processing it, and returns the previous window.

CASE statement. Provides, in C/2, the body of a window procedure. There is one CASE statement for each message type written to take specific actions.

cell. See *character cell.*

CGA. Color graphics adapter.

chained list. A list in which the data elements may be dispersed but in which each data element contains information for locating the next. Synonym for *linked list.*

character. A letter, digit, or other symbol.

character box. In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single character from a character set. See also *character mode.* Contrast with *character cell.*

character cell. The physical, rectangular space in which any single character is displayed on a screen or printer device. Position is addressed by row and column coordinates. Contrast with *character box.*

character code. The means of addressing a character in a character set, sometimes called *code point.*

character mode. The character mode, in conjunction with the font type, determines the extent to which graphics characters are affected by the character box, shear, and angle attributes.

check box. A control window, shaped like a square button on the screen, that can be in a checked or unchecked state. It is used to select one or more items from a list. Contrast with *radio button.*

check mark. The symbol (√) that is used to indicate a selected item on a pull-down.

child window. A window that is positioned relative to another window (either a main window or another child window). Contrast with *parent window.*

choice. An option that can be selected. The choice can be presented as text, as a symbol (number or letter), or as an icon (a pictorial symbol).

class. See *window class.*

class style. The set of properties that apply to every window in a window class.

client area. The area in the center of a window that contains the main information of the window.

clipboard. An area of main storage that can hold data being passed from one Presentation Manager application to another. Various data formats can be stored.

clipping. In computer graphics, removing those parts of a display image that lie outside a given boundary.

clip limits. The area of the paper that can be reached by a printer or plotter.

clipping path. A clipping boundary in world-coordinate space.

code page. An assignment of graphic characters and control-function meanings to all code points.

code point. Synonym for *character code.*

color dithering. A process that simulates a single color on the screen by setting alternate pels, for example, to different colors.

command. (1) The name and parameters associated with an action that can be performed by a program. A command is one form of action request. Users type in the command and enter it. (2) An action users request to interact with the command area.

command area. An area composed of two elements: a command field prompt and a command entry field.

command entry field. An entry field in which users type commands. The entry field is preceded by a command field prompt. These two elements make up the command area.

command line. On a display screen, a display line (usually at the bottom of the screen) in which only commands can be entered.

command prompt. A field prompt showing the location of the command entry field in a panel.

Common Programming Interface. A consistent set of specifications for languages, commands, and calls to

enable applications to be developed across all SAA environments. See also *Systems Application Architecture.*

Common User Access. A set of rules that define the way information is presented on the screen, and the techniques for the user to interact with the information.

control. The means by which an operator gives input to an application. A *choice* corresponds to a control.

Control Panel. In the Presentation Manager, a program used to set up user preferences that act globally across the system.

Control Program. The basic function of OS/2 including DOS emulation and the support for keyboard, mouse, and VIO.

control window. A class of window used to handle a specific kind of user interaction. Radio buttons and check boxes are examples.

correlation. The action of determining which element or object within a picture is at a given position on the display. This follows a *pick* operation.

CPI. Common Programming Interface.

current position. The point from which the next primitive will be drawn.

cursor. A symbol displayed on the screen and associated with an input device. The cursor indicates where input from the device will be placed. Types of cursors include text cursors, graphics cursors, and selection cursors. Contrast with *pointer* and *input focus.*

D

data structure. (ISO) The syntactic structure of symbolic expressions and their storage-allocation characteristics.

DBCS. See *double-byte character set.*

default procedure. Function provided by the Presentation Interface that may be used to process standard messages from dialogs or windows.

default value. A value used when no value is explicitly specified by the user. For example, in the graphics programming interface, the default line type is 'solid'.

Desktop Manager. In the Presentation Manager, a window from which users can start one or more listed programs.

desktop window. The window, corresponding to the physical device, against which all other types of windows are established.

device context. A logical description of a data destination such as memory, metafile, display, printer, or plotter. See also *direct device context, information device context, memory device context, metafile device context, queued device context,* and *screen device context.*

device driver. A file that contains the code needed to attach and use a device such as a display, printer, or plotter.

device space. Coordinate space in which graphics are assembled after all GPI transformations have been applied. Device space is defined in device-specific units.

dialog. The interchange of information between a computer and its user through a sequence of requests by the user and the presentation of responses by the computer.

dialog box. A type of window that contains one or more controls for the formatted display and entry of data. Also known as a *pop-up window.* A modal dialog box is used to implement a pop-up window.

Dialog Box Editor. A what-you-see-is-what-you-get (WYSIWYG) editor that creates dialog boxes for communicating with the application user.

dialog item. A component (for example, a menu or a button) of a dialog box. Dialog items are also used when creating dialog templates.

dialog tag language. A markup language used by the DTL compiler to create dialog objects. It is based on the Standard Generalized Markup Language (SGML).

dialog template. The definition of a dialog box, which contains details of its position, appearance, and window ID, and the window ID of each of its child windows.

direct device context. A logical description of a data destination that is a device other than the screen (for example, a printer or plotter), and where the output is not to go through the spooler. Its purpose is to satisfy queries. See also *device context.*

direct manipulation. The action of using the mouse to move objects around the screen. For example, moving files and directories about in the File Manager.

directory. A type of file containing the names and controlling information for other files or other directories.

display point. Synonym for *pel.*

dithering. The process used in color displays whereby every other pel is set to one color, and the intermediate pels are set to another. Together they produce the effect of a third color at normal viewing distances. This process can only be used on solid areas of color; it does not work on narrow lines, for example.

double-byte character set (DBCS). A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more characters than can be represented by 256 code points, require double-byte character sets. As each character requires two bytes, the entering, displaying, and printing of DBCS characters requires hardware and software that can support DBCS.

dragging. In computer graphics, moving an object on the display screen as if it were attached to the pointer.

drawing chain. See *segment chain.*

drop. To fix the position of an object that is being dragged, by releasing the select button of the pointing device.

DTL. See *dialog tag language.*

dynamic segments. Graphics segments drawn in exclusive-OR mix mode so that they can be moved from one screen position to another without affecting the rest of the displayed picture.

E

EBCDIC. Extended binary-coded decimal interchange code.

EGA. Extended graphics adapter.

element. An entry in a graphics segment that comprises one or more graphics orders and that is addressed by the element pointer.

entry field. A panel element in which users type information. Compare to *selection field.*

entry panel. A defined panel type containing one or more entry fields and protected information such as headings, prompts, and explanatory text.

extended help. A facility that provides users with information about an entire application panel rather than a particular item on the panel.

entry-field control. The means by which the application receives data entered by the user in an entry field. When it has the input focus, it displays a flashing pointer at the position where the next typed character will go.

exit. The action that terminates the current function and returns the user to a higher level function. Repeated exit requests return the user to the point from which all functions provided to the system are accessible. Contrast with *cancel.*

extended-choice selection. A mode that allows the user to select more than one item from a window. Not all windows allow extended choice selection. Contrast with *multiple-choice selection.*

F

field-level help. Information specific to the field on which the cursor is positioned. This help function is "contextual" because it provides information about a specific item as it is currently used; the information is dependent upon the context within the work session.

File Manager. In the Presentation Manager, a program that displays directories and files, and allows various actions on them.

file specification. The full identifier for a file, which includes its file name, extension, path, and drive.

fillet. A curve that is tangential to the end points of two adjoining lines. See also *polyfillet.*

font. A particular size and style of typeface that contains definitions of character sets, marker sets, and pattern sets.

foreground program. The program with which the user is currently interacting. Also known as *interactive program.*

frame. The part of a window that can contain several different visual elements specified by the application, but drawn and controlled by the Presentation Manager. The frame encloses the client area.

frame styles. Different standard window layouts provided by the Presentation Manager.

full-screen application. An application program that occupies the whole screen.

function key. A key that causes a specified sequence of operations to be performed when it is pressed, for example, F1 and Alt-K.

G

Global Descriptor Table (GDT). Defines code and data segments available to all tasks in an application.

glyph. A graphic symbol whose appearance conveys information.

GPI. Graphics Programming Interface. The formally-defined programming language that is between an IBM graphics program and the user of the program. See also *API.*

graphics. A picture defined in terms of graphic primitives and graphics attributes.

graphics attributes. Attributes that apply to graphic primitives. Examples are color, line type, and shading-pattern definition. See also *segment attributes.*

graphics field. The clipping boundary that defines the visible part of the presentation-page contents.

graphics model space. The conceptual coordinate space in which a picture is constructed after any model transforms have been applied. Also known as *model space.*

graphic primitive. A single item of drawn graphics, such as a line, arc, or graphics text string. See also *graphics segment.*

graphics segment. A sequence of related graphic primitives and graphics attributes. See also *graphic primitive.*

graying. The indication that a choice on a pull-down is unavailable.

group. A collection of logically-connected controls. For example, the buttons controlling paper size for a printer. See also *program group.*

H

handle. An identifier that represents an object, such as a device or window, to the Presentation Interface.

hard error. An error condition on a network that requires either that the system be reconfigured, or that the source of the error be removed before the system can resume reliable operation.

heap. An area of free storage available for dynamic allocation by the program. Its size varies, depending on the storage requirements of the program.

help. A function that provides information about a specific field, an application panel, or information about the help facility. It provides field help when the cursor is on a selection or entry field in an application panel or another help panel. It provides information about the application panel, called *extended help,* when the cursor is not in an interactive field.

help index. A facility that allows the user to select topics for which help is available.

help panel. A panel with information to assist users that is displayed in response to a help request from the user.

help window. A Common User Access-defined secondary window that displays information when the user requests help.

hit testing. The means of identifying which window is associated with which input device event.

hook. A mechanism by which procedures are called when certain events occur in the system. For example, the filtering of mouse and keyboard input before it is received by an application program.

hook chain. A sequence of hook procedures that are "chained" together so that each event is passed, in turn, to each procedure in the chain.

hot spot. The part of the pointer that must touch an object before it can be selected. This is usually the tip of the pointer. Contrast with *action point.*

I

icon. A pictorial representation of an item the user can select. Icons can represent items (such as a document file) that the user wants to work on, and actions that the user wants to perform. In the Presentation Manager, icons are used for data objects, system actions, and minimized programs.

icon area. In the Presentation Manager, the area at the bottom of the screen that is normally used to display the icons for minimized programs.

Icon Editor. The Presentation Manager-provided tool for creating icons.

image font. A set of symbols, each of which is described in a rectangular array of pels. Some of the pels in the array are set to produce the image of the symbol. Contrast with *outline font.*

information device context. A logical description of a data destination other than the screen (for example, a printer or plotter), but where no output will occur. Its purpose is to satisfy queries. See also *device context.*

information panel. A defined panel type characterized by a body containing only protected information.

input focus. The area of the screen that will receive input from an input device (typically the keyboard).

input router. OS/2-internal process that removes messages from the system queue.

integer atom. A special kind of **atom** that represents a predefined system constant and carries no storage overhead. For example, names of window classes provided by Presentation Manager are expressed as integer atoms.

interactive graphics. Graphics that can be moved or manipulated by a user at a terminal.

interactive program. A program that is running (active) and is ready to receive (or is receiving) input from the user. Compare with *active program* and contrast with *noninteractive program.*

Also known as a *foreground program.*

interchange file. Data that can be sent from one Presentation Interface application to another.

J

journal. A special-purpose file that is used to record changes made in the system.

K

kanji. A graphic character set used in Japanese ideographic alphabets.

kerning. The design of graphics characters so that their character boxes overlap. Used to space text proportionally.

keys help. A facility that gives users a listing of all the key assignments for the current application.

L

label. In a graphics segment, an identifier of one or more elements that is used when editing the segment.

language support procedure. Function provided by the Presentation Interface for applications that do not, or cannot (as in the case of COBOL and FORTRAN programs), provide their own dialog or window procedures.

LIFO stack. A data stack from which data is retrieved in last-in, first-out order.

linked list. Synonym for *chained list.*

list box. A control window containing a vertical list of selectable descriptions.

list panel. A defined panel type that displays a list of items from which users can select one or more choices and then specify one or more actions to work on those choices.

Local Descriptor Table (LDT). Defines code and data segments specific to a single task.

M

main window. The window that is positioned relative to the desktop window.

marker box. In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single marker from a marker set.

marker symbol. A symbol centered on a point. Graphs and charts can use marker symbols to indicate the plotted points.

maximize. A window-sizing action that makes the window the largest size possible.

media window. The part of the physical device (display, printer, or plotter) on which a picture is presented.

memory device context. A logical description of a data destination that is a memory bit map. See also *device context.*

menu. A type of panel that consists of one or more selection fields. Also called a *menu panel.*

message. 1. In Presentation Manager, a packet of data used for communication between the Presentation Interface and windowed applications.

2. In a user interface, information not requested by users but presented to users by the computer in response to a user action or internal process. Messages on status, problems, or user actions from a computer application should be distinguished from a "message" or note sent to users by other users over a communications link.

message filter. The means of selecting which messages from a specific window will be handled by the application.

message queue. A sequenced collection of messages to be read by the application.

metafile. The generic name for the definition of the contents of a picture. Metafiles are used to allow pictures to be used by other applications.

metafile device context. A logical description of a data destination that is a metafile, which is used for graphics interchange. See also *device context.*

metalanguage. A language used to specify another language. In this publication, the data types are described using a metalanguage so as to make the descriptions independent of any one computer language.

mickey. A unit of measurement for physical mouse motion whose value depends on the mouse device driver currently loaded.

micro presentation space. A graphics presentation space in which a restricted set of the GPI function calls is available.

minimize. A window-sizing action that makes the window the smallest size possible. In the Presentation Manager, minimized windows are represented by icons.

mix. An attribute that determines how the foreground of a graphic primitive is combined with the existing color of graphics output. Also known as *foreground mix*. Contrast with *background mix*.

mixed character string. A string containing a mixture of one-byte and *kanji* or *hangeul* (two-byte) characters.

mnemonic. A method of selecting an item on a pull-down by means of typing the highlighted letter in the menu item.

modal dialog box. The type of control that allows the operator to perform input operations on only the current dialog box or one of its child windows. Also known as a *serial dialog box*. Contrast with *parallel dialog box*.

modeless dialog box. The type of control that allows the operator to perform input operations on any of the application's windows. Also known as a *parallel dialog box*. Contrast with *modal dialog box*.

model space. See *graphics model space*.

mouse. A hand-held device that is moved around to position the pointer on the screen.

multiple-choice selection. A mode that allows users to select any number of choices, including none at all. See also *check box*. Contrast with *extended-choice selection*.

multitasking. The concurrent processing of applications or parts of applications. A running application and its data are protected from other concurrently running applications.

N

named pipe. A named object that provides client-to-server, server-to-client, or duplex communication between unrelated processes. Contrast with *unnamed pipe*.

noninteractive program. A program that is running (active) but is not ready to receive input from the user. Compare with *active program*, and contrast with *interactive program*.

nonretained graphics. Graphic primitives that are not remembered by the Presentation Interface once they have been drawn. Contrast with *retained graphics*.

null-terminated string. A string of (n + 1) characters where the (n + 1)th character is the 'null' character (X'00'), and is used to represent an n-character string with implicit length. Also known as 'zero-terminated' string and 'ASCIIZ' string.

O

object window. A window that does not have a parent, but which may have child windows. An object window cannot be presented on a device.

open. To start working with a file, directory, or other object.

outline font. A set of symbols, each of which is created as a series of lines and curves. Contrast with *image font*.

output area. The area of the output device within which the picture is to be displayed, printed, or plotted.

owner window. A window into which specific events that occur in another (owned) window are reported.

P

page viewport. A boundary in device coordinates that defines the area of the output device in which graphics are to be displayed. The presentation-page contents are transformed automatically to the page viewport in device space.

paint. The action of drawing or redrawing the contents of a window.

panel. A particular arrangement of information grouped together for presentation to the user in a window.

panel area. An area within a panel that contains related information. The three major panel areas defined by the Common User Access are the action bar, the function key area, and the panel body.

panel body. The portion of a panel not occupied by the action bar, function key area, title or scroll bars. The panel body may contain protected information, selection fields, and entry fields. The layout and content of the panel body determine the panel type.

panel body area. The part of a window not occupied by the action bar or function key area. The panel body area may contain information, selection fields, and entry fields. Also known as *client area*.

panel body area separator. A line or color boundary that provides users with a visual distinction between two adjacent areas of a panel.

panel definition. A description of the contents and characteristics of a panel. Thus, a panel definition is the application developer's mechanism for predefining the format to be presented to users in a window.

panel ID. A panel element located in the upper left-hand corner of a panel body that identifies that particular panel within the application.

panel title. A panel element that identifies the information in the panel.

paper size. The size of paper, defined in either standard U.S. or European names (for example, A, B, A4), and measured in inches or millimeters respectively.

parallel dialog box. See *modeless dialog box.*

parent window. The window relative to which one or more child windows are positioned. Contrast with *child window.*

pel. The smallest area of a display screen capable of being addressed and switched between visible and invisible states. Synonymous with *display point, pixel,* and *picture element.*

pick. To select part of a displayed object using the pointer.

picture chain. See *segment chain.*

picture element. Synonym for *pel.*

pipe. See *named pipe, unnamed pipe.*

pixel. Synonym for *pel.*

plotter. An output device that uses pens to draw its output on paper or transparency foils.

pointer. The symbol displayed on the screen that is moved by a pointing device, such as a *mouse.* The pointer is used to point at items that users can select. Contrast with *cursor.*

pointing device. A device (such as a mouse) used to move a pointer on the screen.

pointings. Pairs of x-y coordinates produced by an operator defining positions on a screen with a pointing device, such as a *mouse.*

polyfillet. A curve based on a sequence of lines. It is tangential to the end points of the first and last lines, and tangential also to the midpoints of all other lines. See also *fillet.*

polyline. A sequence of adjoining lines.

pop. To retrieve an item from a last-in-first-out stack of items. Contrast with *push.*

pop-up window. A window that appears on top of another window in a dialog. Each pop-up window must be completed before returning to the underlying window.

Presentation Manager. The OS/2 Control Program plus the visual component that presents, in windows, a graphics-based interface to applications and files installed and running in OS/2.

presentation page. The coordinate space in which a picture is assembled for display.

presentation space (PS). Contains the device-independent definition of a picture.

primary window. The window in which the main dialog between users and the application takes place. In a multi-programming environment, each application starts in its own primary window. The primary window remains for the duration of the application, although the panel displayed will change as the user's dialog moves forward. See also *secondary window.*

primitive. See *graphic primitive.*

primitive attribute. A specifiable characteristic of a graphic primitive. See *graphics attributes.*

print job. The result of sending a document or picture to be printed.

Print Manager. In the Presentation Manager, the part of the spooler that manages the spooling process. It also allows users to view print queues and to manipulate print jobs.

process. An instance of an executing application and the resources it is using.

program details. Information about a program that is specified in the Desktop Manager window and is used when the program is started.

program group. In the Presentation Manager, several programs that can be acted upon as a single entity.

program name. The full file specification of a program. Contrast with *program title.*

program title. The name of a program as it is listed in the Desktop Manager window. Contrast with *program name.*

pull-down. An extension of the *action bar* that displays a list of choices available for a selected action bar choice. After users select an action bar choice, the pull-down appears with the list of choices. Additional *pop-up windows* may appear from pull-down choices to further extend the actions available to users.

push. To add an item to a last-in-first-out stack of items. Contrast with *pop*.

pushbutton. A control window, shaped like a rounded-corner rectangle and containing text, that invokes an immediate action, such as 'enter' or 'cancel'.

Q

queue. A list of print jobs waiting to be printed.

queued device context. A logical description of a data destination (for example, a printer or plotter) where the output is to go through the spooler. See also *device context*.

R

radio button. A control window, shaped like a round button on the screen, that can be in a checked or unchecked state. It is used to select a single item from list. Contrast with *check box*.

reentrant. The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

reference phrase. A word or phrase that is emphasized in a device-dependent manner in order to inform the user that additional information for the word or phrase is available.

reference phrase help. Provides help information on a selectable phrase.

refresh. To update a window, with changed information, to its current status.

region. A clipping boundary in device space.

resource. The means of providing extra information used in the definition of a window. A resource can contain definitions of fonts, templates, accelerators, and mnemonics; the definitions are held in a resource file.

restore. To return a window to its original size or position following a sizing or moving action.

retained graphics. Graphic primitives that are remembered by the Presentation Interface after they have been drawn. Contrast with *nonretained graphics*.

reverse video. A form of alphanumeric highlighting for a character, field, or cursor, in which its color is exchanged with that of its background. For example, changing a red character on a black background to a black character on a red background.

RGB. Red-green-blue. For example "RGB display".

roman. Relating to a type style with upright characters.

S

screen. The physical surface of a workstation or terminal upon which information is presented to users.

screen device context. A logical description of a data destination that is a particular window on the screen. See also *device context*.

scroll bar. A control window, horizontally or vertically aligned, that allows the user to scroll additional data into an associated panel area.

scrollable entry field. An entry field larger than the visible field.

scrollable selection field. A selection field that contains more choices than are visible.

scrolling. Moving a display image vertically or horizontally in a manner such that new data appears at one edge, as existing data disappears at the opposite edge.

secondary window. A type of window associated with the primary window in a dialog. A secondary window begins a secondary and parallel dialog that runs at the same time as the primary dialog.

segment. See *graphics segment*.

segment attributes. Attributes that apply to the segment as an entity, as opposed to the individual primitives within the segment. For example, the visibility or detectability of a segment.

segment chain. All segments in a graphics presentation space that are defined with the 'chained' attribute. Synonymous with *picture chain*.

segment priority. The order in which segments are drawn.

segment store. An area in a normal graphics presentation space where retained graphics segments are stored.

select. To mark or choose an item. Note that *select* means to mark or type in a choice on the screen; *enter* means to send all selected choices to the computer for processing.

select button. The button on a pointing device, such as a mouse, that is pressed to select a menu choice. Also known as button 1.

selection cursor. A type of cursor used to indicate the choice or entry field users want to interact with. It is represented by highlighting the item it is currently positioned on.

selection field. A field containing a list of choices from which the user can select one or more.

semaphore. An object used by multi-threaded applications for signaling purposes and for controlling access to serially reusable resources.

separator. See *panel body area separator.*

serial dialog box. See *modal dialog box.*

serially reusable resource (SRR). A logical resource or object that can be accessed by only one task at a time.

session. A routing mechanism for user interaction via the console.

shadow box. The area on the screen that follows mouse movements and shows what shape the window will take if the mouse button is released.

shear. The tilt of graphics text when each character leans to the left or right while retaining a horizontal baseline.

shutdown. In the Desktop Manager, the procedure required before the computer is switched off to ensure that data is not lost.

sibling windows. Child windows that have the same parent window.

slider box. An area on the scroll bar that indicates the size and position of the visible information in a panel area in relation to the information available. Also known as *thumb mark.*

spline. A sequence of one or more Bezier curves.

spooler. A program that intercepts the data going to printer devices and writes it to disk. The data is printed or plotted when it is complete, and the required device is available. The spooler prevents output from different sources being intermixed.

standard window. A collection of windows that form a panel.

static control. The means by which the application presents descriptive information (for example, headings and descriptors) to the user. The user cannot change this information.

style. See *window style.*

suballocation. The allocation of a part of one extent for occupancy by elements of a component other than the one occupying the remainder of the extent.

switch. An action that moves the input focus from one area to another. This can be within the same window or from one window to another.

switch list. See Task List.

symbolic identifier. A text string that equates to an integer value in an include file, that is used to identify a programming object.

System Menu. In the Presentation Manager, the pull-down in the top left corner of a window that allows it to be moved and sized with the keyboard.

system queue. This is the master queue for all pointer device or keyboard events.

Systems Application Architecture. A formal set of rules that enables applications to be run without modification in different computer environments.

T

tag. A markup language word and its attributes that are entered in the source file to identify parts of a panel or other dialog objects.

Task List. In the Presentation Manager, the list of programs that are active. The list can be used to switch to a program and to stop programs.

template. An ASCII-text definition of an action bar and pull-down menu, held in a resource file, or as a data structure in program memory.

text. Characters or symbols.

text cursor. A symbol displayed in an entry field that indicates where typed input will appear.

text window. Also known as the VIO window. The environment in which OS/2 runs AVIO applications.

thread. A unit of execution within a process.

thumb mark. The portion of the scroll bar that describes the range and properties of the data that is currently visible in a window. Also known as a *slider box.*

tilde. A mark used to denote the character that is to be used as a mnemonic when selecting text items within a menu.

title bar. The area at the top of a window that contains the window title. The title bar is highlighted when that window has the input focus. Contrast with *panel title.*

transform. (1) The action of modifying a picture by scaling, shearing, reflecting, rotating, or translating. (2) The object that performs or defines such a modification; also referred to as a *transformation.*

Tree. In the Presentation Manager, the window in the File Manager that shows the organization of drives and directories.

U

unnamed pipe. A circular buffer, created in memory, used by related processes to communicate with one another. Contrast with *named pipe.*

update region. A system-provided area of dynamic storage containing one or more (not necessarily contiguous) rectangular areas of a window, that are visually invalid or incorrect, and therefore in need of repainting.

User Shell. A component of OS/2 that uses a graphics-based, windowed interface to allow the user to manage applications and files installed and running under OS/2.

V

vector font. A set of symbols, each of which is created as a series of lines and curves. Contrast with *image font* and *outline font.*

VGA. Video graphics array.

viewing pipeline. The series of transformations applied to a graphic object to map the object to the device on which it is to be presented.

viewing window. Clipping boundary that defines the visible part of model space.

VIO. Video Input/Output.

virtual memory (VM). Addressable space that is apparent to the user as the processor storage space, but not having a fixed physical location.

visible region. A window's presentation space, clipped to the boundary of the window and the boundaries of any overlying window.

W

wild-card character. The global file-name characters ? or *.

window. A rectangular area of the screen through which a panel or portion of a panel is displayed. A window can be smaller than or equal in size to the screen. Windows can overlap on the screen and give the appearance of one window being on top of another.

window class. The grouping of windows whose processing needs conform to the services provided by one window procedure.

window coordinates. The means by which a window position or size is defined; measured in device units, or *pels.*

window procedure. Code that is activated in response to a message.

window rectangle. The means by which the size and position of a window is described in relation to the desktop window.

window style. The set of properties that influence how events related to a particular window will be processed.

workstation. A display screen together with attachments such as a keyboard, a local copy device, or a tablet.

world coordinates. Application-convenient coordinates used for drawing graphics.

world-coordinate space. Coordinate space in which graphics are defined before transformations are applied.

WYSIWYG. What You See Is What You Get. A capability that enables text to be displayed on a screen in the same way that it will be formatted on a printer.

Z

z-order. The order in which sibling windows are presented. The topmost sibling window obscures any portion of the siblings that it overlaps; the same effect occurs down through the order of lower sibling windows.

zooming. In graphics applications, the process of increasing or decreasing the size of picture.

Index

A

a-space of a character 23-3, 23-7
accelerators 5-13
access modes of files 33-12
accessing DOS disks 33-3
accessing HPFS disks from DOS 33-3
accessing OS/2 FAT disks from DOS 33-3
action bar 5-8
 definition of X-1
action, definition of X-1
active window 3-5
adapter state, setting by full-screen applications 30-9
adapter, video 30-5
address 36-13, 37-2, 37-3
address of procedure 34-17
ADVISE transaction 29-7
alignment, text 23-13
allocate memory 32-1
alphanumeric data
 printing 36-6, 36-12
alphanumeric video output
 accessing logical video buffer 30-3
 creating a Vio pop-up 30-6
 creating Vio presentation space 30-1
 defining logical fonts 30-2
 enabling ANSI support 30-5
 manipulating the cursor 30-6
 querying adapter type 30-5
 scrolling text 30-7
 setting code pages 30-5
 writing strings 30-4
alternate mode, area primitive 20-11
anchor block handle 4-5, 11-10
ANSI video support 30-5
API
 family 1-4, 1-7
 function call types 1-6
 function calls used by DOS-mode applications 1-8
 function calls used by full-screen applications 1-8
 function calls used by PM applications 1-7
 function calls used by text-windowed applications 1-7
application interface to FSD 33-6
application termination 4-7
application type, determining the 34-11
application-driven dynamic data exchange 29-1
applications, sample
 CLIPBRD 28-3
 DIALOG1 9-1—9-9
 DIALOG2 5-3—5-5, 10-1—10-22
 GRAPHIC1 19-7, 22-15—22-18
 HELLO1 4-1—4-12
 HELLO2 6-1—6-9
applications, sample *(continued)*
 IMAGE 13-10—13-14
arbitrary units 19-4
arc parameters 20-13
arc primitive 20-12
 attributes 20-18
 circle 20-13
 current arc parameters 20-13
 direction of drawing 20-13
 ellipse 20-13
 fillet 20-16
 full arc 20-14
 partial arc 20-14
 spline 20-17
 3-point arc 20-16
archive file attribute, changing 33-13
ARE parameter 37-7
area primitive 20-10
 attributes 20-12
 calls valid in B-1
 in alternate mode 20-11
 in winding mode 20-11
area-fill pattern 20-12
artlink tag (IPF) H-3
artwork tag (IPF) H-5
ASCII mode 31-2
ASN.1 33-19
aspect ratio 19-4
Assembler
 API calls in 1-6
ASSOCTABLE EA, EAF_DEFAULTOWNER 33-21
ASSOCTABLE EA, EAF_REUSEICON 33-22
ASSOCTABLE EA, EAF_UNCHANGEABLE 33-21
ASSOCTABLE extended attribute 33-21
asynchronous drawing thread 18-2
asynchronous notification 35-6
asynchronous painting 3-20
atom management 13-21
atom table 13-21
attaching file systems 33-2, 33-5
attribute modes 22-11
attributes
 color 21-5
 of arcs 20-18
 of areas 20-12
 of character strings 23-9
 of images 20-4
 of lines 20-3
 of markers 20-3
attributes of files 33-12
average character width 23-3
AVIO cell sizes 30-2

B

b-space of a character 23-3, 23-7
background mix 21-7
background of graphic primitives 21-5
background thread 34-11
beep
 turning on and off 14-1
Bezier spline 20-17
bibliography vii
binary mode 31-2
bit maps 24-1
 as area-fill patterns 23-16
 bit-map plane 24-1
 conversion between color and monochrome 24-6
 creating 24-2
 deleting 24-4
 drawing 24-2
 editing 8-1
 for redrawing the screen contents 24-1
 format 24-1
 giving to other processes 24-4
 Icon Editor 24-3
 information header 24-2
 information table 24-4
 loading 24-4
 memory device context 24-2
 mix values 24-5
 pel manipulation 24-7
 printing 36-11
 saving 24-4
 size 24-5
 standard formats 24-1
 use to copy the screen 36-10
 when to use 24-1
books, list of vii
bootable HPFS 33-5
borders, sizing 5-15
boundary determination 26-8
box primitive 20-2
 direction of drawing 20-11
 filled 20-2
 with rounded corners 20-2
break character 23-14
break statement 3-8
byte pipes 35-12

C

c-space of a character 23-3, 23-7
cache buffers, saving 33-32
cached micro presentation space 19-1, 19-6
CACHE.EXE, HPFS lazy-write utility 33-5
caching data, HPFS 33-4
called segments 22-6
calling convention 1-9
Canadian-French code page (CP863) 39-3
case sensitivity, file name 33-7
case statement 3-8
caution tag (IPF) H-8
cell, character 30-1
cgraphic tag (IPF) H-9
chained attribute of graphics segments 22-6
character angle 23-11
character attributes 23-8
character box 23-9
character box, default 23-10
character cell 30-1
character device monitors 38-1
character devices, file I/O 33-1
character direction 23-13
character fonts 23-1
 See also fonts
 average character width 23-3
 code pages 23-6
 current 23-2
 device 23-3, 23-5
 example code
 defining a logical font 23-16
 facename 23-3
 family name 23-3
 generic 23-3
 installing 23-2
 kerning 23-4, 23-7
 locally defined 23-1
 logical 23-2
 match value 23-3
 maximum baseline extent 23-3
 metrics 23-4
 modes 23-4, 23-8
 physical 23-2
 private 23-2
 proportional spacing 23-7
 public 23-2
 resolution 23-6
 selecting 23-2
 simulation 23-3
 system-provided 23-1
character mode 23-4, 23-8
character reference point 23-7, 23-11
character set 39-8
character shape 39-2
character shear 23-12
character string
 attributes 23-9
 character angle 23-11
 character box 23-9
 character direction 23-13
 character mode 23-8
 character shear 23-12
character strings
 reading from keyboard 31-2
 replicating to LVB 30-5
 saving from LVB 30-5
 writing to LVB 30-4

character width, fonts 8-8
character-string primitive 23-8
characters (IPF) I-1
CHCP command 39-3
check box 7-8
child process 34-9
child session 34-11
child windows 3-1
CHKDSK 33-3, 33-5
circle primitive 20-13, 20-14
class style
 CS_PUBLIC 13-9
client window 6-1
clip paths
 defining in segments 26-1
Clipboard 1-3, 28-1
 application view of 28-1
 calls 28-6
 copy 28-1
 cut 28-1
 for sharing bit maps 24-4
 for sharing metafiles 27-5
 GpiQueryMetaFileBits 27-5
 GpiQueryMetaFileLength 27-5
 GpiSetMetaFileBits 27-5
 owner 28-3
 paste 28-1
CLIPBRD sample program 28-3
clipping 26-1
 boundary 26-1, 26-6
 graphics field 25-7, 26-3
 implementation 26-1
 path 26-1
 selecting 26-2
 region 26-4, 26-6
 rules 26-1
 viewing window 25-5, 26-2
clock device, file I/O 33-1
CLOCK$ 33-10
close-segment processing 22-3
closing a queue 35-10
COBOL 3-8
 API calls in 1-6
COBOL PM programs 2-2
 bit management 11-19
 constants 11-6, 11-11
 constructing and decomposing 4-byte integers 11-15
 DLG1COB 11-1
 DLG2COB 11-1, 12-1
 message processing 11-15
 message-processing loop 11-12
 PM limitations 3-11
 program structure 11-4
 structures 11-6, 12-15, 12-35
COBOL program structure 11-4
COBOL/2 constants files 2-2
COBOL/2 structures 2-3
code page 39-1
 API function calls 39-12
 Canadian-French code page (CP863) 39-3
 character set 39-8
 character shape 39-2
 CHCP 39-3, 39-4, 39-6
 code page management 39-2
 code-page dependent information 39-7
 code-page operation 39-5
 code-page preparation 39-3
 code-page switching example 39-7
 default 39-4
 differences between 437 and 850 39-8
 EBCDIC code pages 39-3
 folding 39-11
 function calls 39-4, 39-12
 keyboard scan-code input 39-7
 keyboard scan-code output 39-5
 KEYBOARD.DCP file 39-3
 languages supported by CP850 39-8
 monocasing 39-11
 multilingual code page (CP850) 39-3, 39-8
 Nordic code page (CP865) 39-3
 Portugese code page (CP860) 39-3
 product design considerations 39-10
 selecting a language version of an application 39-1
 selection by COUNTRY.SYS file 39-3
 selection for graphics characters 23-6
 special considerations and limitations 39-10
 spooled printer output 39-5
 supported devices 39-10
 switching 39-7
 tag 39-5
 utility and command output 39-5
 U.S. code page (CP437) 39-3, 39-8
 video output 39-5
code page, setting for the keyboard 31-4
code page, setting video 30-5
codepage, extended attributes 33-18
COL parameter 37-8
color 21-1
 attribute 21-5
 available colors 21-4
 background 21-6
 background mix 21-7
 background of graphic primitives 21-5
 dithering 21-4
 foreground 21-6
 foreground mix 21-6
 foreground of graphic primitives 21-5
 index 21-3
 intensity 21-1
 leave-alone mix 21-9
 OR mix 21-8
 output on monochrome devices 21-9
 overpaint mix 21-7
 planes 21-1
 querying system colors 14-7

color *(continued)*
specifying for primitives 21-6
tables 21-1
default logical 21-2
defining 21-3
formats 21-3
index mode 21-3
investigating 21-4
loaded 21-3
logical 21-1
physical 21-2
realizing 21-4
RGB mode 21-3
XOR mix 21-8
color printing 37-8
color tag (IPF) H-10
COM device, file I/O 33-1
combination box 7-17
combining the transformation matrixes 25-14
combo box 7-17
command area, definition of X-2
command entry field, definition of X-2
command prompt, definition of X-2
command, definition of X-2
comment control word (IPF) H-2
COMMENT extended attribute 33-20
comment parameter 37-3
Common User Access 1-1, 1-3, 5-9, 7-4, 7-9
benefits 3-23
character and graphics applications 3-23
consistency 3-23
introduction 3-22
pop-up window 3-23, 9-8
primary window 3-23
secondary window 3-23
selection in list box 10-17
comp parameter 37-2
compatibility boxes, hardware system limits G-2
COM1,COM2,COM3 33-10
CON 33-10
concatenation, transformations 25-17
concurrent running 1-4
CONFIG.SYS 33-2, 39-3, 39-4, 39-10
CODEPAGE statement 39-3, 39-4
COUNTRY statement 39-4
DEVINFO statement 39-3, 39-4, 39-10
console device, file I/O 33-1
Console keyboard and screen devices 33-10
constants
in COBOL 11-11
in FORTRAN 11-11
control panel
querying and changing functions 14-1
control window class 3-13
control windows 7-7
check boxes 7-8
entry fields 7-12
list box 7-10
control windows *(continued)*
pushbuttons 7-8
radio buttons 7-7
scroll bar 7-18
controls for user dialog 7-1
Convert Picture File utility 27-7
converting PC-DOS interrupts to OS/2 function calls F-1
coordinate positions 19-3
coordinate range 19-4
coordinate spaces
device 25-8
model 25-2
presentation-page 25-4
summary 25-10
viewing pipeline 25-1
world 25-1
coordinates, window 3-2
COP parameter 37-7
correlation 22-8, 22-12
flag 22-14
function calls 22-13
hit 22-14
nonretained graphics 22-14
on called segments 22-14
pick aperture 22-13
pick tag 22-13
retained graphics 22-13
count parameter 37-1
COUNTRY.SYS file 39-3
critical EAs 33-26, 33-27
CS_ class styles 4-5
CS_ class style, see window class 3-14
CS_PUBLIC 13-9
currency 18-2
current arc parameters 20-13
default values 20-13
defining a circle 20-13
defining an ellipse 20-13
current attribute value 20-1
current font 23-2
current position 19-3
current transformation 25-1
cursor, manipulating 30-6, G-2
customizing the user interface 14-1—14-7
C/2
API calls in 1-6
C/2 files 2-1

D

DASD flag, setting 33-12
data exchange
application-driven
See Dynamic Data Exchange (DDE)
user-driven
See clipboard

data stream 29-7
data types, registering 13-4
datatype parameter 37-2
data, holding 32-1
DBCS (double-byte character set)
 entry field 7-13
DDE (Dynamic Data Exchange) 29-1
debugging process 34-17
decreasing segment size 32-5
default code page 39-4
default device transformation 25-8
default logical color table 21-2
default viewing transformation 25-4, 25-6
DEFAULTICON 33-20
definition list tags (IPF) H-13
 dd (definition description) H-11
 ddhd H-12
 dl (definition list) H-13
 dt (definition term) H-15
 dthd H-16
deleting a file 33-14
deleting a queue 35-10
deny none access 33-12
deny read access 33-12
deny read/write access 33-12
deny write access 33-12
descriptor entry 32-1
Desktop Manager
 See also installed program list
 querying and changing functions 14-1
detaching file systems 33-5
detectability attribute of graphics segments 22-8, 22-12
device calls 1-7, 1-8
 DevEscape codes
 DEVESC_ABORTDOC 36-12
 DEVESC_DRAFTMODE 36-9
 DEVESC_ENDDOC 36-6, 36-8, 36-9
 DEVESC_GETSCALINGFACTOR 36-4
 DEVESC_NEWFRAME 36-6, 36-8, 36-9
 DEVESC_QUERYESCSUPPORT 36-4, 36-9
 DEVESC_QUERYVIOCELLSIZES 36-4
 DEVESC_RAWDATA 36-6
 DEVESC_STARTDOC 36-6, 36-7, 36-9, 37-3
 long names
 DevCloseDC 19-3, 27-3, 36-6, 36-8, 36-10, 36-11
 DevEscape 27-6, 36-4
 DevOpenDC 24-2, 27-2, 37-1—37-3
 DevPostDeviceModes 37-4, 37-5
 DevQueryCaps 21-4, 21-9, 23-6, 23-7, 23-10, 36-4, 37-6
 DevQueryDeviceNames 36-3
 DevQueryHardcopyCaps 19-4, 36-4, 37-5, 37-6
 short names
 WDCLS 19-3, 27-3, 36-6, 36-8, 36-10, 36-11
 WDESC 27-6, 36-4
 WDOPEN 24-2, 27-2, 37-1—37-3
 WDPDM 37-4, 37-5
 WDQCAP 21-4, 21-9, 23-6, 23-7, 23-10, 36-4, 37-6
device calls *(continued)*
 short names *(continued)*
 WDQDN 36-3
 WDQHC 19-4, 36-4, 37-5, 37-6
device context
 associating with a presentation space 19-2
 closing 19-3
 creating 37-1
 direct 36-13, 37-1
 handle 19-2, 37-1
 information 37-1, 37-5
 memory 24-2, 36-10, 37-1
 metafile 3-22, 27-2
 printer 37-1, 37-10
 purpose 19-2
 queued 36-2, 36-6, 37-1
 types 3-21, 37-1
device drivers 36-5, 38-1
device fonts 23-3
device I/O DOS calls 38-1
device names reserved by OS/2 33-10
device space 25-8
 limits 25-9
device transformation 19-5, 25-8
device type 33-6
device-transformation
 matrix 25-19
DEVOPENDATA 37-2, 37-4
DevOpenDC parameters 37-1
dialog box 3-24
 check box 7-8
 controls 7-1, 7-23
 creation 7-22
 definition 3-24, 7-21
 entry field 7-12
 grid 7-23
 layout 7-21
 list box 7-10
 modal 7-2
 application modal 7-2
 system-modal 7-2
 modeless 7-2
 nonreentrant languages 10-22
 order groups 7-25
 procedure 9-8
 pushbutton 7-8
 radio buttons 7-7
 scroll bar 7-18
 template 7-26
 testing 7-25
 units 7-27
 use 3-24
 user input 9-8
Dialog Box Editor 7-21, 7-27
Dialog Manager applications 1-2, 1-3, 1-4, 1-7
dialog procedure 7-1, 9-8, 10-5
dialog tag language
 definition of X-3

dialog template 7-1, 7-21, 7-26
DIALOG1 sample application 9-1—9-9
DIALOG2 sample application 10-1—10-22
direct device context 36-13, 37-1
directives
 in resource script file 8-6
directories
 See subdirectories
discardable segments 32-2
discardable segments, accessing 32-4
disk
 accessing as a single file 33-12
 DOS disk accessing 33-3
 ensuring order of data written 33-15
 handling critical errors 33-15
 verifying data written 33-16
disk error reporting 33-12
disk partition size G-2
diskette drives, hardware system limits G-2
dispatchable unit of execution 34-1
display mode, setting for full-screen applications 30-7
displays, hardware system limits G-2
dithering, definition of X-3
DLGTEMPLATE statement 7-26
DLG1COB.COB dialog box sample application 11-1
DLG1COB.FOR dialog box sample application 11-1
DOS accessing OS/2 FAT disks 33-3
DOS accessing OS/2 HPFS disks 33-3
DOS calls 1-7
 DosAllocHuge 32-2
 DosAllocSeg 13-18, 32-1
 DosAllocShrSeg 32-3
 DosBufReset 33-32
 DosCallBack 34-1
 DosCallNmPipe 35-15
 DosChDir 33-15
 DosChgFilePtr 33-32
 DosCLIAccess. 38-2
 DosClose 33-15, 35-14, 36-7
 DosCloseQueue 35-10
 DosCloseSem 35-2
 DosConnectNmPipe 35-13
 DosCopy 33-13
 DosCreateCSAlias 32-4
 DosCreateQueue 35-9
 DosCreateThread 34-1
 DosCWait 34-9
 DosDelete 33-14
 DosDevIOCtl 33-6
 DosDisConnectNmPipe 35-13
 DosDupHandle 33-14
 DosEditName 33-10, 33-11
 DosEnterCritSec 34-3
 DosEnumAttribute 33-23
 DosErrClass A-1
 DosError A-1
 DosExecPgm 34-9
 DosExit 34-3, 34-9
DOS calls *(continued)*
 DosExitCritSec 34-3
 DosExitList 34-9
 DosFileIO 33-32
 DosFileLocks 33-32
 DosFindClose 33-29
 DosFindFirst 33-28
 DosFindFirst2 33-23
 DosFindNext 33-29
 DosFlagProcess 35-6
 DosFreeModule 34-17
 DosFreeResource 32-8
 DosFreeSeg 32-4, 32-7
 DosFSAttach 33-6
 DosFSCtl 33-6
 DosFSRamSemClear 35-4
 DosFSRamSemRequest 35-4
 DosGetCp 39-5
 DosGetCtryInfo 39-7
 DosGetEnv 34-9
 DosGetHugeShift 32-2
 DosGetInfoSeg 34-14
 DosGetMessage 39-2
 DosGetModHandle 34-16
 DosGetModName 34-16
 DosGetPID 34-11
 DosGetPPID 34-11
 DosGetProcAddr 34-17
 DosGetPrty 34-10
 DosGetResource 32-7
 DosGetResource2 32-8
 DosGetSeg 32-3
 DosGetShrSeg 32-3
 DosGiveSeg 32-3
 DosInsMessage 39-2
 DosKillProcess 34-9
 DosLoadModule 8-10, 13-9, 34-16
 DosLockSeg 32-4
 DosMakeNmPipe 33-1, 35-12
 DosMakePipe 33-1, 35-8
 DosMemAvail 32-3
 DosMkDir 33-13
 DosMkDir2 33-13
 DosMonClose 38-2
 DosMonOpen 38-2
 DosMonRead 38-2
 DosMonReg 38-2
 DosMonWrite 38-2
 DosMove 33-13
 DosMuxSemWait 35-4, 35-6
 DosNewSize 33-13
 DosOpen 33-11, 35-13, 36-7
 DosOpenQueue 35-9
 DosOpenSem 35-2
 DosOpen2 33-12
 DosPeekNmPipe 35-15
 DosPeekQueue 35-9
 DosPortAccess 38-2

DOS calls *(continued)*
DosPtrace 34-17
DosPurgeQueue 35-10
DosPutMessage 39-2
DosQAppType 34-11
DosQCurDir 33-15
DosQFHandState 33-14
DosQFileInfo 33-23
DosQFileMode 33-13
DosQFSAttach 33-6
DosQHandType 33-15
DosQNmPHandState 35-16
DosQNmPipeSemState 35-17
DosQPathInfo 33-23
DosQSysInfo 33-11
DosQueryQueue 35-10
DosQVerify 33-16
DosRead 33-31, 35-14
DosReadAsync 33-31, 35-14
DosReadQueue 35-9
DosReallocHuge 32-4, 32-5
DosReallocSeg 32-4, 32-5
DosResumeThread 34-3
DosRmDir 33-15
DosR2StackRealloc 34-1
DosScanEnv 33-29
DosSearchPath 33-29
DosSelectSession 34-13
DosSemClear 35-4
DosSemRequest 35-1, 35-4
DosSemSet 35-4
DosSemSetWait 35-6
DosSendSignal 35-7
DosSetCp 39-5
DosSetFHandState 33-14
DosSetFileInfo 33-23
DosSetFileMode 33-13
DosSetMaxFH 33-15
DosSetNmPHandState 35-16
DosSetNmPipeSem 35-16
DosSetPathInfo 33-23
DosSetProcCp 39-5
DosSetPrty 34-10
DosSetSession 34-13
DosSetSigHandler 35-6, A-1
DosSetVec 35-6, A-1
DosSetVerify 33-16
DosShutDown 33-6
DosSizeSeg 32-7
DosSleep 34-14
DosSMRegisterDD 34-13
DosStartSession 8-4, 34-11
DosStopSession 34-13
DosSubAlloc 32-5
DosSubFree 32-7
DosSubSet 32-5
DosSuspendThread 34-3
DosTimerAsync 34-14
DOS calls *(continued)*
DosTimerStart 34-14
DosTimerStop 34-14
DosTransactNmPipe 35-15
DosUnlockSeg 32-4
DosWrite 33-31, 35-14, 36-7
DosWriteAsync 33-31, 35-14
DosWriteQueue 35-9
DOS FAT file system 33-2, 33-3
DOS mode 1-4
DOS mode, accessing HPFS disks in 33-4
DOS-mode applications 1-4
DosClose 33-1
DosFreeSeg 32-7
draw-and-retain-mode graphics 22-1
draw-mode graphics 22-1
drawing controls
boundary-data flag 26-8
correlate flag 22-14
display flag 22-10
draw-dynamic-segments flag 22-10
erase-before-draw flag 22-10
drawing modes, graphics 22-1
DRIVDATA 37-2, 37-5
driverdata 37-2, 37-4
drivername 37-2
drives 33-2
dual boot function 33-5
duplicating a handle to file, pipe, or device 33-14
dynamic attribute of graphics segments 22-8
Dynamic Data Exchange (DDE)
implementation rules 29-5
Dynamic Data Exchange (DDE) 29-1
ADVISE transaction 29-7
conversation 29-3, 29-6
data stream 29-7
EXECUTE command string, syntax 29-10
message set 29-5
WM_DDE_ACK 29-7
WM_DDE_DATA 29-6, 29-8
WM_DDE_EXECUTE 29-8
WM_DDE_INITIATE 29-6
WM_DDE_POKE 29-7
WM_DDE_TERMINATE 29-9
WM_DDE_UNADVISE 29-8
one-time data transfer 29-6
POKE transaction 29-7
protocol definition 29-5
REQUEST transaction 29-6
shared memory object rules 29-10
synchronization rules 29-9
system topic 29-10
terminology and basic rules 29-3
the workings of the protocol 29-4
uses for 29-3
Dynamic Link Libraries (DLL)
creating 1-6

dynamic link library
public window class, registering 13-9
window procedure 13-9
dynamic link module 34-16
dynamic linking 1-6
address of procedure 34-17
at run-time 34-16
definition of 34-15
file name retrieval 34-16
freeing the reference 34-17
load-on-demand segments 34-16
loading the module 34-16
LOADONCALL 34-15
long file names 33-8
placeholders 34-15
PRELOAD 34-15
Presentation Manager application 32-7
resource file 8-10
resource segments, accessing 32-8
resource segments, freeing 32-8
dynamic segments
at associate time 22-10
definition 22-8
drawing 22-8, 22-10, 25-3
in hard-copy output 22-10
in metafiles 27-2
mix mode 21-8
position in the segment chain 22-8

E

EA data types
ASN.1 33-19
include data 33-19
multi-valued, multi-typed 33-18
multi-valued, single-type 33-18
summary 33-18
EAF_DEFAULTOWNER, ASSOCTABLE EA 33-21
EAF_REUSEICON, ASSOCTABLE EA 33-22
EAF_UNCHANGEABLE, ASSOCTABLE EA 33-21
EAOPs 33-24
EAUTIL 33-28
EBCDIC code pages 39-3
echo mode 31-2
edit modes, graphics segments 22-4
editing characters 33-10
Editor
Dialog Box 7-27
Font 8-8, 23-1
Icon 8-1, 24-3
element pointer, graphics 22-3
element type, graphics 22-4
elements
deleting 22-5
editing 22-4
inserting 22-4
replacing 22-4
elements of a graphics segment 22-3
calls valid in B-1
ellipse primitive 20-13, 20-14
ellipsis (...) 3-24
emptying the queue 35-10
entry field 7-12
multiline entry field 7-12
non-display for password entry 13-3
single line entry field 7-12
entry field styles
ES_ANY 7-12
ES_CENTER 7-12
ES_DBCS 7-12
ES_LEFT 7-12
ES_MARGIN 7-12
ES_MIXED 7-12
ES_PICTUREMASK 7-12
ES_READONLY 7-12
ES_RIGHT 7-12
ES_SBCS 7-12
ES_UNREADABLE 7-12
MLS_BORDER 7-14
MLS_HSCROLL 7-14
MLS_IGNORETAB 7-14
MLS_READONLY 7-14
MLS_VSCROLL 7-14
MLS_WORDWRAP 7-14
entry field, definition of X-4
entry panel
definition of X-4
equations, transformation 25-11
error API calls A-1
ERROR device 33-1
ES_SBCS 7-13
ES_, see entry field styles 7-12
example code
defining a logical font 23-16
printer device context 37-10
printing using a separate thread 36-14
printing using device calls 36-16
example tag (IPF) H-59
exception error, 80286 machine A-1
exclusive-OR mix 21-8
extended attributes
data types 33-18
nam 33-17
naming conventions 33-17
size 33-16
use 33-16
value 33-17
extended help
definition of X-4
extended partitions 33-5
external events 35-6
external leading 23-14

F

facename of a graphics-character font 23-3
family API 1-4, 1-7
family applications 1-4, 1-8
family name of a graphics-character font 23-3
fast-chaining attribute of graphics segments 22-7
FAT file system, DOS 33-2, 33-3
FAT file system, OS/2 33-2, 33-3
FATTRS (font attributes) 23-2
FDISK 33-2
FEAList 33-24
FEAs 33-24
FEA_NEEDEA 33-27
FID_ values 5-1
field-level help
 definition of X-4
figure tags (IPF)
 fig (figure) H-17
 figcap (figure caption) H-18
file access 33-12
file allocation table, FAT 33-2
file archives 33-12
file attributes 33-12
file handle state flags 33-12
file information, level 1 33-16
file I/O 33-1
file name format, FAT 33-6
file name format, FAT, DOS 33-7
file name format, IFS 33-6
file names, long 33-8
file pointer 33-30
file sharing 33-12
file system
 DOS accessing OS/2 FAT disks 33-3
 DOS accessing OS/2 HPFS disks 33-3
 drivers 33-2
 dual boot function 33-5
 extended attributes mechanism 33-12
 high performance file system 33-4
 installable file system 33-3
file system driver 33-2
FileAttribute parameter, DosOpen 33-12
files
 changing attributes of 33-13
 changing the size of 33-13
 closing 33-15
 creating 33-11
 deleting 33-14
 duplicating handles of 33-14
 locking regions of 33-32
 moving the pointer 33-30
 moving to another subdirectory 33-13
 naming conventions 33-6
 opening existing 33-11
 reading from 33-31
 renaming 33-13
 replacing 33-11
files *(continued)*
 saving buffers 33-32
 setting maximum handle amount 33-15
 sharing 33-12
 writing to 33-31
FileSize parameter, DosOpen 33-12
fillet primitive 20-16
filtering messages 13-19
filtering, messages 3-7
FIT parameter 37-7
fixed disks, hardware system limits G-2
flags 35-6
flushing cache buffers 33-32
folding 39-11
font attributes (FATTRS) 23-2
Font Editor 8-8, 23-1
font file for full-screen applications 30-10
font metrics 23-4
font-use indicators 23-4
fonts 8-8, 23-1
 See also character fonts
 adding public fonts 14-1, 14-2
 creating 23-1
 current 23-2
 example code
 defining a logical font 23-16
 image 23-1
 logical 23-2
 defining 23-2
 outline 23-1
 physical 23-2
 private 23-2
 public 23-2, 36-9
 raster 23-1
 selecting 23-2
 system 23-1
 transformable 23-1
 using public fonts 14-1, 14-2
 vector 23-1
fonts, loading by full-screen applications 30-9
footnotes (IPF) H-20
foreground mix 21-6
foreground of graphic primitives 21-5
foreground session 1-4
foreground session, temporary 30-6
foreground thread 34-11
form names 36-1, 37-3, 37-4
FORMAT 33-3, 33-5
formatting a drive 33-3
forms
 descriptions 37-5
 parameter 37-4
 selection 37-4
FORTRAN 3-8
 API calls in 1-6
FORTRAN PM programs 2-2
 bit management 11-19
 constants 11-6, 11-11

FORTRAN PM programs *(continued)*
constructing and decomposing 4-byte integers 11-15
DLG1FOR 11-1
DLG2FOR 11-1
message processing 11-15
message-processing loop 11-12
PM limitations 3-11
structures 11-7, 12-15, 12-35
FORTRAN/2 constants files 2-2
FORTRAN/2 structures 2-3
frame creation flags
FCF_ACCELTABLE 5-6
FCF_BORDER 5-15
FCF_ICON 5-6, 5-8
FCF_MENU 5-6, 5-8
FCF_SHELLPOSITION 5-5
FCF_SIZEBORDER 5-15
FCF_STANDARD 5-1, 5-5, 5-6, 5-7, 5-8
FCF_TASKLIST 5-8
FCF_TITLEBAR 5-7
FRAMECDATA structure 5-6
freeing handles 33-1
freeing memory 32-7
FSD name 33-6
FSD (file system driver) 33-2
full arc primitive 20-14
full-screen applications 1-4
function key, definition of X-4
function prototypes 4-4

G

GEAList 33-24
GEAs 33-24
generic fonts 23-3
getting handles 33-1
global characters 33-11
global descriptor tables, system limits G-1
global file name characters 33-9
global sharing of segment 32-3
GPI (Graphics Programming Interface) 18-1
graphic data
printing 36-7
graphics
arbitrary units 19-4
attributes 22-2, C-1
color 21-5
mix 21-6
modes 22-11
resetting process C-1
saving before changing 22-11
boundary determination 26-8
character fonts 23-1
facename 23-4
local identifier (lcid) 23-5
match value 23-4
clearing the screen before drawing 22-10
color 21-1
See also color
graphics *(continued)*
color *(continued)*
on monochrome devices 21-9
coordinate spaces 25-1
coordinates 19-3
correlation 22-12
draw-and-retain 22-1
draw-mode 22-1
drawing modes 22-1
elements
classifying 22-4
locating 22-3, 22-4
example code
associating the presentation space 19-7
boundary determination 26-9
changing the segment transformation 25-20
defining a logical font 23-16
using bit maps 24-7
field 25-7, 26-1, 26-3
interactive 25-1
interchange requirements 27-6
metric units 19-4
moving around the screen 22-8
nonretained 22-1
orders 22-2, 22-4, D-1
decoding D-1
naming conventions D-1
pels 19-3
presentation page 19-1
See also presentation page
redefining 22-12
presentation space 19-1, 36-7
See also presentation space
restoring 22-12
saving 22-12
primitives 20-1, 22-9
See also primitives
query calls 18-2
retain-mode 22-1
retained 22-1
sample programs 18-2
scrolling 25-6
segments 22-1
See also segment
called from other segments 22-11
closing 22-2
contents of 22-2
labels 22-4, 22-5
stop-draw condition 18-2
text 23-1
formatting 23-13
horizontal formatting 23-13
vertical formatting 23-14
text, printing 36-8
viewing pipeline 25-1, 25-10
viewing window 25-5
zooming 25-6

graphics calls
long names
GpiAssociate 19-2, 36-7, 36-8, 36-9, 36-10, 36-11, 37-1
GpiBeginArea 20-11
GpiBeginElement 22-3
GpiBeginPath 20-18
GpiBitBlt 2-9, 24-3, 24-5, 36-10, 36-11
GpiBox 20-2, 20-11, 36-8
GpiCallSegmentMatrix 22-6, 25-3
GpiCharString 23-8, 36-7, 36-8, 36-9
GpiCharStringAt 4-8, 6-9, 23-8, 36-8, 36-9
GpiCharStringPos 36-9
GpiCharStringPosAt 36-9
GpiCloseFigure 20-18
GpiCloseSegment 22-2, 22-4
GpiCombineRegion 26-4
GpiConvert 25-9, 25-11, 26-7
GpiCopyMetaFile 27-5
GpiCorrelateChain 22-13
GpiCorrelateFrom 22-13
GpiCorrelateSegment 22-13
GpiCreateBitmap 24-2, 36-10
GpiCreateLogColorTable 21-3, 21-4
GpiCreateLogFont 23-2, 36-9, 39-7
GpiCreatePS 3-19, 19-1, 36-7, 36-10, 36-11
GpiCreateRegion 26-4
GpiDeleteBitmap 24-4, 36-11
GpiDeleteElement 22-5
GpiDeleteElementRange 22-5
GpiDeleteElementsBetweenLabels 22-5
GpiDeleteMetaFile 27-5
GpiDeleteSegment 22-5
GpiDeleteSegments 22-5
GpiDeleteSetId 23-6, 23-16
GpiDestroyPS 19-3, 36-8, 36-10, 36-11
GpiDestroyRegion 26-6
GpiDrawChain 22-6, 22-9, 36-8
GpiDrawDynamics 22-10, 25-3, 27-2
GpiDrawFrom 22-9, 36-7
GpiDrawSegment 22-9, 36-7
GpiElement 22-4
GpiEndArea 20-11
GpiEndElement 22-3
GpiEndPath 20-18
GpiEqualRegion 26-6
GpiErase 21-9, 22-10
GpiErrorSegmentData 22-9
GpiFillPath 20-19
GpiFullArc 20-14
GpiGetData 22-4, 22-5
GpiImage 20-4
GpiLabel 22-4
GpiLine 20-1, 36-7
GpiLoadBitmap 24-4
GpiLoadFonts 8-9, 23-2
GpiLoadMetaFile 27-3
GpiMarker 20-3
GpiModifyPath 20-20

graphics calls *(continued)*
long names *(continued)*
GpiMove 22-11, 36-9
GpiOffsetElementPointer 22-3, 22-4
GpiOffsetRegion 26-6
GpiOpenSegment 22-2, 22-3, 22-4
GpiOutlinePath 20-20
GpiPaintRegion 26-6
GpiPartialArc 20-14
GpiPlayMetaFile 27-3, 36-8
GpiPointArc 20-16
GpiPolyFillet 20-16
GpiPolyFilletSharp 20-16
GpiPolyLine 20-1, 22-2, 36-8
GpiPolyMarker 20-3
GpiPolySpline 20-17
GpiPop 22-11
GpiPtInRegion 26-6
GpiPtVisible 26-7
GpiPutData 22-4, 22-5, 22-6
GpiQueryAttrs 20-18
GpiQueryBackMix 21-9
GpiQueryBitmapBits 24-4
GpiQueryBitmapHandle 23-16
GpiQueryBitmapParameters 24-4
GpiQueryBoundaryData 26-8
GpiQueryCharDirection 23-13
GpiQueryCharStringPos 23-13
GpiQueryCharStringPosAt 23-13
GpiQueryColorData 21-4, 21-5
GpiQueryColorIndex 21-5
GpiQueryCp 23-6, 39-7
GpiQueryDefCharBox 23-10
GpiQueryDeviceBitmapFormats 24-2
GpiQueryEditMode 22-5
GpiQueryElement 22-3
GpiQueryElementType 22-4
GpiQueryFontFileDescriptions 23-2
GpiQueryFontMetrics 23-5, 23-6
GpiQueryFonts 23-2, 36-9
GpiQueryKerningPairs 23-7
GpiQueryMetaFileBits 27-5
GpiQueryMetaFileLength 27-5
GpiQueryMix 21-9
GpiQueryNearestColor 21-5, 21-10
GpiQueryNumberSetIds 23-16
GpiQueryPageViewport 25-9
GpiQueryPel 24-7
GpiQueryRealColors 21-5
GpiQueryRegionBox 26-6
GpiQueryRegionRects 26-6
GpiQueryRGBColor 21-5
GpiQuerySegmentNames 22-2
GpiQuerySegmentPriority 22-7
GpiQueryWidthTable 23-7
GpiRealizeColorTable 21-4
GpiRectInRegion 26-6
GpiRectVisible 26-7
GpiRemoveDynamics 22-10, 25-3, 27-2

graphics calls *(continued)*
long names *(continued)*
GpiResetBoundaryData 26-8
GpiResetPS 22-5, 22-12
GpiRestorePS 22-12
GpiRotate 25-18
GpiSaveMetaFile 27-3, 27-5
GpiSavePS 22-12
GpiScale 25-19
GpiSetArcParams 20-13
GpiSetAttrMode 22-11
GpiSetAttrs 20-18, 21-6, 21-9
GpiSetBackColor 6-9, 21-6
GpiSetBackMix 6-9, 21-9
GpiSetBitmap 24-2, 36-10, 36-11
GpiSetBitmapBits 24-4
GpiSetBitmapId 23-16
GpiSetCharAngle 23-11
GpiSetCharBox 23-9
GpiSetCharDirection 23-13
GpiSetCharMode 23-8
GpiSetCharSet 23-2, 23-5, 36-9
GpiSetCharShear 23-12
GpiSetClipPath 26-2
GpiSetClipRegion 26-6
GpiSetColor 4-8, 6-9, 21-6
GpiSetCp 23-6, 39-3, 39-7
GpiSetCurrentPosition 19-3, 20-11, 36-8, 36-9
GpiSetDefAttrs 20-1
GpiSetDefaultViewMatrix 25-6
GpiSetDrawControl 22-10, 22-14, 26-8
GpiSetDrawingMode 22-1
GpiSetEditMode 22-5
GpiSetElementPointer 22-3
GpiSetElementPointerAtLabel 22-4
GpiSetGraphicsField 25-7, 26-3
GpiSetInitialSegmentAttrs 22-9
GpiSetLineEnd 20-19
GpiSetLineJoin 20-19
GpiSetLineType 20-3
GpiSetLineWidthGeom 20-19
GpiSetMarker 20-3
GpiSetMarkerBox 20-4
GpiSetMarkerSet 23-16
GpiSetMetaFileBits 27-5
GpiSetMix 21-9, 22-8
GpiSetModelTransformMatrix 25-3
GpiSetPageViewport 19-6, 25-8
GpiSetPattern 20-12
GpiSetPatternRefPoint 20-12
GpiSetPatternSet 23-16
GpiSetPel 24-7
GpiSetPickAperturePosition 22-14
GpiSetPickApertureSize 22-13
GpiSetPS 22-5, 22-12, 27-4
GpiSetRegion 26-4
GpiSetSegmentAttrs 22-9
GpiSetSegmentPriority 22-7
GpiSetSegmentTransformMatrix 25-3

graphics calls *(continued)*
long names *(continued)*
GpiSetStopDraw 18-2, 19-1, 27-2
GpiSetTag 22-13
GpiSetViewingLimits 25-5, 26-2
GpiSetViewingTransformMatrix 25-5
GpiStrokePath 20-19
GpiTranslate 25-18
GpiUnloadFonts 23-2, 23-6
GpiUnrealizeColorTable 21-4
GpiWCBitBlt 24-3, 24-7
short names
GIATR 20-18, 21-6, 21-9
GIBB 2-9, 24-3, 24-5, 36-10, 36-11
GIBOX 20-2, 20-11, 36-8
GICA 23-11
GICALM 22-6, 25-3
GICB 23-9
GICH 23-12
GICHAR 4-8, 6-9, 23-8, 36-8, 36-9
GICHPP 36-9
GICHPR 36-9
GICONV 25-9, 25-11, 26-7
GICORF 22-13
GICORS 22-13
GICP 19-3, 20-11, 36-8, 36-9
GICSEG 22-13
GIDATR 20-1
GIFARC 20-14
GIFLD 25-7, 26-3
GIGARC 20-16
GIGLW 20-19
GILINE 20-1, 36-7
GIMARK 20-3
GIMB 20-4
GIMOVE 22-11, 36-9
GIMRKS 20-3
GIOREG 26-6
GIPAP 22-14
GIPARC 20-14
GIPCRT 3-19, 19-1, 36-7, 36-10, 36-11
GIPDEL 19-3, 36-8, 36-10, 36-11
GIPEL 24-7
GIPFLS 20-16
GIPFLT 20-16
GIPIR 26-6
GIPLNE 20-1, 22-2, 36-8
GIPRP 20-12
GIPS 22-5, 22-12, 27-4
GIPSPL 20-17
GIPV 19-6, 25-8
GIPVIS 26-7
GIQATR 20-18
GIQBD 26-8
GIQCEL 23-10
GIQCPO 23-13
GIQCRO 23-13
GIQPEL 24-7
GIQPV 25-9

graphics calls *(continued)*
short names *(continued)*
GIQRB 26-6
GIQRR 26-6
GIREG 26-4
GIREGN 26-4
GIRIR 26-6
GIRVIS 26-7
GISAP 20-13
GISCTM 25-3
GISDVM 25-6
GISPAS 22-13
GISTFM 25-3
GISVL 25-5, 26-2
GISVTM 25-5
GIWBB 24-3, 24-7
GSAM 22-11
GSAREA 20-11
GSASS 19-2, 36-7, 36-8, 36-9, 36-10, 36-11, 37-1
GSBBIT 24-4
GSBCOL 6-9, 21-6
GSBMID 23-16
GSBMIX 6-9, 21-9
GSCBM 24-2, 36-10
GSCD 23-13
GSCHAP 23-8, 36-7, 36-8, 36-9
GSCLCT 21-3, 21-4
GSCM 23-8
GSCOL 4-8, 6-9, 21-6
GSCPG 23-6, 39-3, 39-7
GSCREG 26-4
GSCRLF 23-2, 36-9, 39-7
GSCS 23-2, 23-5, 36-9
GSDBM 24-4, 36-11
GSDC 22-10, 22-14, 26-8
GSDCHN 22-6, 22-9, 36-8
GSDDYN 22-10, 25-3, 27-2
GSDELS 22-5
GSDF 22-9, 36-7
GSDLE 22-5
GSDLEB 22-5
GSDLER 22-5
GSDM 22-1
GSDREG 26-6
GSDSEG 22-9, 36-7
GSEDM 22-5
GSELEM 22-3
GSENDA 20-11
GSENDE 22-3
GSENDP 20-18
GSEP 22-3
GSEPLB 22-4
GSEREG 26-6
GSERSE 21-9, 22-10
GSESD 22-9
GSFCLS 20-18
GSFILP 20-19
GSFLO 8-9, 23-2
GSFUL 23-2, 23-6

graphics calls *(continued)*
short names *(continued)*
GSGET 22-4, 22-5
GSIMG 20-4
GSINLB 22-4
GSLBM 24-4
GSLE 20-19
GSLJ 20-19
GSLOAD 27-3, 36-8
GSLT 20-3
GSMCPY 27-5
GSMDEL 27-5
GSMFB 27-5
GSMGD 27-5
GSMIX 21-9, 22-8
GSMLOD 27-3
GSMODP 20-20
GSMS 20-3
GSMSAV 27-3, 27-5
GSMSS 23-16
GSOEP 22-3, 22-4
GSOUTP 20-20
GSPAT 20-12
GSPATH 20-18
GSPATS 23-16
GSPOP 22-11
GSPREG 26-6
GSPUT 22-4, 22-5, 22-6
GSQBB 24-4
GSQBMX 21-9
GSQBTH 23-16
GSQBTP 24-4
GSQCD 23-13
GSQCOD 21-4, 21-5
GSQCOI 21-5
GSQCPG 23-6, 39-7
GSQDBF 24-2
GSQEDM 22-5
GSQETS 22-4
GSQF 23-2, 36-9
GSQFD 23-2
GSQFM 23-5, 23-6
GSQFWT 23-7
GSQKPR 23-7
GSQMFL 27-5
GSQMIX 21-9
GSQNAM 22-2
GSQNC 21-5, 21-10
GSQNSS 23-16
GSQPRI 22-7
GSQRC 21-5
GSQRGB 21-5
GSRBD 26-8
GSRCT 21-4
GSRDYN 22-10, 25-3, 27-2
GSROT 25-18
GSRPS 22-5, 22-12
GSRSPS 22-12
GSRSS 23-6, 23-16

graphics calls *(continued)*
short names *(continued)*
GSSATI 22-9
GSSATS 22-9
GSSBM 24-2, 36-10, 36-11
GSSCAL 25-19
GSSCLP 26-2
GSSCLR 26-6
GSSCLS 22-2, 22-4
GSSDEL 22-5
GSSDW 18-2, 19-1, 27-2
GSSEG 22-2, 22-3, 22-4
GSSPRI 22-7
GSSPS 22-12
GSSTRP 20-19
GSTAG 22-13
GSUCT 21-4
GSXLAT 25-18
Graphics Programming Interface (GPI) 18-1
GRAPHIC1 sample application 19-7, 22-15—22-18
grid, dialog box 7-23

H

handles, file system 33-1, 33-2
handles, freeing 33-1
handles, getting 33-1
handle, determining type 33-15
hard error processing A-1
hard-copy output 36-1
hardware system limits G-2
header files 2-1
header file, graphics orders D-1
headings (IPF) H-21
heap handle 13-22
heap management function calls 13-24
heap manager 13-22
heap sizes, system limits G-2
heaps, moveable 13-23
HELLO1 sample application 4-1, 4-12
HELLO2 sample application 6-1—6-9
Help
definition of X-5
implementation 5-15
supplying help 5-15
help facility
See Information Presentation Facility
help facility calls
WinAssociateHelpInstance 16-2
WinCreateHelpInstance 16-1
WinCreateHelpTable 16-10
WinDestroyHelpInstance 16-2
WinLoadHelpTable 16-11
WinQueryHelpInstance 16-13
help hook 13-19
help index
definition of X-5
Help Manager utilities
GML to IPF conversion utility, definition of 2-14
IPFC 2-14
help panel 8-5
definition of X-5
help table 8-5
help window
definition of X-5
hidden file attribute, changing 33-13
Hidden files 33-12
hide tag (IPF) H-24
hierarchy of windows 3-1
High Performance File System
bootable disk 33-5
CACHE utility 33-5
caching data 33-4
description of 33-4
dual-boot systems 33-5
file name format 33-7
file system utilities 33-5
lazy writing 33-4
non-bootable disk 33-5
path names 33-4
system files 33-5
highlighting (IPF) H-26
HISTORY extended attribute 33-20
HK_, see hooks 13-19
holding data 32-1
hooks 13-19
help 5-15
HK_CODEPAGECHANGE 13-19
HK_HELP 13-19
HK_INPUT 13-19
HK_JOURNALPLAYBACK 13-19
HK_JOURNALRECORD 13-19
HK_MSGFILTER 13-19
HK_SENDMSG 13-19
horizontal scan lines per cursor G-2
HPFS disks, accessing 33-3
HPFS file system driver, HPFS.IFS 33-5
HPFS file system utilities, UHPFS.DLL 33-5
HPFS lazy-write utility, CACHE.EXE 33-5
HPFS (see High Performance File System)
HPFS.IFS, HPFS file system driver 33-5
HSTRUCT data type 12-35
hypertext 17-13

I

icon
associating with text-windowed and full-screen programs 8-4
device independent 8-3
device specific 8-3
editing 8-1
hotspot 8-3
Icon Editor 8-1
draw straight facility 8-2

Icon Editor *(continued)*
grid 8-2
palette 8-1
pensize 8-2
ICON extended attribute 33-20
identity transformation 25-1, 25-15
IFS 33-2
IFS= 33-2
image fonts 23-1, 23-8
image primitive 20-4
attributes 20-4
IMAGE sample application 13-10—13-14
imbed control word (IPF) H-31
include file (.H)
for resource script file 8-6
in resource script files 7-27
include files 2-1, 4-4
increasing segment size 32-5
independent process 34-9
independent session 34-11
index mode, color tables 21-3
index tags (IPF)
icmd (index command) H-30
isyn (index synonyms) H-32
i1 and i2 (index entries) H-28
information device context 37-1, 37-5
information header, bit-map 24-2
information panel
definition of X-5
Information Presentation Facility
application coding
associating a help instance 16-2
creating a help instance 16-1
creating a new help table 16-10
destroying a help instance 16-2
HELP1 Sample Program 16-3
initialization structure 16-1
loading a new help table 16-11
message processing 16-6
querying a help instance 16-13
resource definition 16-11
setting the active window 16-2
table structures 16-10
artlink H-3
artwork H-5
caution H-8
cgraphic (character graphic) H-9
color H-10
comment H-2
dd (definition description) H-11
ddhd H-12
dl (definition list) H-13
dt (definition term) H-15
dthd H-16
fig (figure) H-17
figcap H-18
fn (footnote) H-20
hide H-24
Information Presentation Facility *(continued)*
hp (highlight phrase) H-26
hx (headings) H-21
icmd (index command) H-30
imbed H-31
isyn (index synonyms) H-32
i1, i2 (index entries) H-28
li (list item) H-34
libraries 17-4
compiling 17-17
file structure for 17-4
location of 17-19
National Language Support 17-18
lines H-35
link H-37
lm H-40
lp (list part) H-42
note H-43
nt (note) H-44
ol (ordered list) H-45
p (paragraph) H-46
parml (parameter list) H-47
pd (parameter description) H-49
pt (parameter term) H-50
rm H-51
sl (simple list) H-53
symbols I-1
tag language markup
characteristics of 17-2
defining a panel 17-2
definition of 17-1
linking from a graphic 17-16
linking from a text phrase 17-13
text presentation 17-6
ul (unordered list) H-55
user interface characteristics
application action bar 15-1
help action bar 15-3
help pull-down 15-6
list box windows 15-3
main window 15-2
options pull-down 15-5
services pull-down 15-3
text window 15-2
userdoc H-57
warning H-58
xmp (example) H-59
information table, bit-map 24-4
inheritance of handles 33-12
initialization file 37-3
initialization file (OS2.INI) 13-9
initialization files 14-1—14-4
contents 14-1
creating default versions 14-2
customizing 14-1
function calls 14-2
initializing for suballocation 32-5

INPUT device 33-1
input focus 31-1
input hook 13-19
input process 3-15
input queue, windows 3-6
input router 3-15
installable file system 33-2
installable file systems 33-2
installed program list 14-3, 14-4
 function calls 14-4
 querying and changing 14-3
instance transformation 25-3
integer atom 13-21
integer atom, system limits G-2
intensity, color 21-1
interactive graphics 22-12, 25-1
Interactive Presentation Facility H-1
interchange requirements, metafiles 27-6
interface design
 consistency 3-23
internal leading 23-15
interprocess communication 1-1
interprocess communication mechanisms
 named pipes 35-11
 queues 35-8
 semaphores 35-1
 shared memory segments 32-3
 unnamed pipes 35-7
IOCtl DOS calls 38-1
IOPL code segments 38-2
IOPL privilege levels 34-1

J

job properties 37-4
journal play back hook 13-19
 MsgFilterHook 13-19
journal record hook 13-19

K

Kbd Calls 1-7
 KbdCharIn 31-2
 KbdDeRegister 31-5
 KbdFlushBuffer 31-3
 KbdFreeFocus 31-3
 KbdGetCP 31-4
 KbdGetFocus 31-1, 31-3
 KbdGetHWid 31-4
 KbdOpen 31-3
 KbdPeek 31-3
 KbdRegister 31-4
 KbdSetCP 31-4, 39-5
 KbdSetCustXt 31-4
 KbdSetStatus 31-3
 KbdStringIn 31-2
 KbdSynch 31-5
 KbdXlate 31-3
KBD$ 33-10
kernel device driver 36-5
kerning, graphics fonts 23-7
Keyboard device 33-10
keyboard device, file I/O 33-1
keyboard hardware id 31-4
keyboard input 3-16
keyboard input by text-windowed and full-screen applications
 clearing the input buffer 31-3
 creating a secondary keyboard 31-3
 input focus 31-1
 loading a keyboard translation table 31-4
 peeking at a data record 31-3
 reading a character string 31-2
 reading a data record 31-2
 replacing keyboard calls 31-4
 setting the keyboard code page 31-4
 translating a data record 31-3
keyboard messages 3-3
keyboards, hardware system limits G-2
KEYBOARD.DCP file 39-3
keyboard, control of state 13-18
KEYPHRASE extended attribute 33-20
keys help
 definition of X-6

L

labels, graphics segments 22-4, 22-5
language-support window procedure 3-11
languages supported 1-8
lazy writing, HPFS 33-4
lcid (logical-font identifier) 23-5, 27-4
LDT (see Local Descriptor Table)
leave-alone mix 21-9
left margin (IPF) H-40
level 1 file information 33-16
libraries, IPF
 See Information Presentation Facility
libraries, system limits G-1
LIFO (last-in, first-out) stack
 for storing presentation-space variables 22-12
 for storing primitive attributes 22-11
limits
 device space 25-9
 global descriptor tables G-1
 hardware G-2
 libraries G-1
 list box items 7-10
 local descriptor tables G-1
 memory resources G-1
 named shared segments G-1
 open file handles G-3
 pipes G-3
 Presentation Manager G-2
 processes G-3
 queues G-3

limits *(continued)*
recovering from G-1
segments G-1
sessions G-2
system semaphores G-3
system signals G-3
threads G-3
windows 3-24
world-coordinate space 19-4, 25-9
limits in Presentation Manager 13-1
line increment of a font 23-14
line primitive 20-1
attributes 20-3
line types 20-3
line width
fixed 20-3
geometric 20-19
linear equations, transformation 25-11
lines tag (IPF) H-35
link tag (IPF) H-37
linking, dynamic
See dynamic linking
list box
description 7-10
limits 7-10
list panel
definition of X-6
list tags (IPF)
See also definition list and parameter list
li (list item) H-34
lp (list part) H-42
ol (ordered list) H-45
sl (simple list) H-53
ul (unordered list) H-55
load list DLL 13-9
load-on-demand segments 34-16
load-time dynamic linking 34-15
loaded color tables 21-3
loading FSDs 33-2
LOADONCALL segment 34-15
local descriptor table system limits G-1
Local Descriptor Table (LDT) 32-1
local file system 33-5
local identifier, graphics font 23-5
local sharing of segments 32-3
locking/unlocking regions in a file 33-13
locking/unlocking shared segments 32-4
logical color table 21-1
logical color table, system limits for G-2
logical drives, hardware system limits G-2
logical file devices 33-1
logical fonts 23-2
defining 23-2
logical fonts, defining for Vio presentation space 30-2
logical fonts, system limits G-2
logical storage device 33-2
logo
adjusting display time 14-1, 14-2
LogoDisplayTime 14-2
long file names
DOS mode 33-8
dynamic link library 33-8
moving files 33-8
NEWFILES 33-8
OS/2 mode 33-8
programming considerations 33-9
restrictions 33-8
8.3 file name format 33-7
LPT1, LPT2, LPT3 33-10
LRU, least recently used data 33-4

M

machine exception error, 80286 A-1
Macro Assembler/2 files 2-1
macros
extraction 4-9, 10-14
macros, for extracting data from message parameters 3-3
MAKEINI.EXE 14-2
managing help requests
See Information Presentation Facility
manuals, list of vii
MAP parameter 37-8
marker primitive 20-3
attributes 20-3
marker sets 23-15
marking EAs 33-27
markup for help panels
See tag language markup (IPF)
match value, graphics fonts 23-3
matrix
device-transformation 25-19
equations for transformation 25-14
rotation 25-14
scaling 25-14
translation 25-14
maximum baseline extent 23-3, 23-14
maximum path length 33-11
memory
overcommitment 1-6
memory device context 24-2, 37-1
memory management
See also segment of memory
allocating memory 32-1
avoiding large allocations 32-3
child process 32-3
deallocating segment 32-7
definition of 32-1
determining availability 32-3
discardable segments 32-4
file name retrieval 34-16
freeing memory 32-7
global sharing of segment 32-3
heap manager 13-22
increasing/decreasing segments 32-5

memory management *(continued)*
load-on-demand segments 34-16
local sharing of segments 32-3
memory suballocation 32-4
multiple segments, allocating 32-2
multiple segments, reallocating 32-5
reallocating a segment 32-5
reallocating memory 32-4
reset segment for suballocation 32-6
run-time dynamic linking 34-16
segments up to 64KB 32-1
shared segment 32-1
shift count 32-2
size of allocated segment 32-7
size of unallocated block 32-3
suballocating segment 32-5
system limits G-1
unlocking discardable segment 32-4
usage count 32-7
window words 13-14
64KB needed by memory 32-2
memory management function calls
DosAllocHuge 32-2
DosAllocSeg 32-1
DosAllocShrSeg 32-3
DosCreateCSAlias 32-4
DosFreeSeg 32-7
DosGetHugeShift 32-2
DosLockSeg 32-4
DosMemAvail 32-3
DosReallocHuge 32-5
DosReallocSeg 32-5
DosSizeSeg 32-7
DosSubAlloc 32-5
DosSubFree 32-7
DosSubSet 32-5
DosUnlockSeg 32-4
Memory Sub-Allocation Package (MSP)
drawbacks 13-24
memory suballocation package 32-4
menu control messages 5-11
menu item attributes 5-10
menu item styles 5-10
menu mnemonics 5-13
menu template 5-9
message
definition of X-6
message box 3-24, 7-19
application-modal 7-20
icons 7-20
modality 7-20
movable 7-20
pushbuttons 7-19
system-modal 7-20
message file 39-2
message loop 4-7
message pipes 35-12
message preparation utilities
make message file (MKMSGF) 2-13
message bind (MSGBIND) 2-13
message processing, IPF 16-6
message queue 36-12
message queue creation 4-5
message tables 8-4
messages
button 7-9
default processing 4-9
filtering 3-7, 13-19
function of 3-5
in resources 8-4
keyboard 3-3
list box 7-11
menu control 5-11
mouse 3-4
multiline entry field 7-15
national language support 8-4
priority 3-17
scroll bar control 5-18
scrollbar notification 5-16
translating 8-4
translating for full-screen applications 39-2
translating for Presentation Manager 39-1
user 7-19
WMRUM, register user 13-4
WM_ACTIVATE 3-11
WM_BUTTON1DOWN 4-8
WM_CHAR 4-8
WM_CLOSE 4-9, 6-9, 9-8, 10-5, 11-15, 12-4
WM_COMMAND 6-8, 9-8, 10-2, 10-7, 10-10, 10-13, 10-17, 11-15, 11-19, 12-4
WM_CONTROL 3-11, 10-10, 10-13, 10-17, 10-19
WM_CREATE 6-8, 9-8, 10-2
WM_ERASEBACKGROUND 6-9, 10-2
WM_HELP 12-4
WM_INITDLG 10-10, 10-13, 10-17
WM_PACTIVATE 3-11
WM_PAINT 3-11, 4-8, 6-9, 9-8, 10-2
WM_PCONTROL 3-11
WM_PPAINT 3-11, 11-15, 12-4
WM_PRESPARAMCHANGED 13-3
WM_PSETFOCUS 3-11
WM_PSIZE 3-11
WM_PSYSCOLORCHANGE 3-11
WM_SAVEAPPLICATION 14-5
WM_SETFOCUS 3-11, 6-8
WM_SIZE 3-11
WM_SUBSTITUTESTRING 8-4
WM_SYSCOLORCHANGE 3-11
WM_USER application-defined 13-3
meta characters 33-11
editing characters 33-10
search characters 33-10
metafile device context 27-2
metafiles
contents 27-1, 27-2

metafiles *(continued)*
copying 27-5
creating 27-2
creating from PIF files 27-7
deleting 27-5
editing 27-5
effect of current drawing mode 27-2, 27-3
handle 27-3
interchange requirements 27-6
loading from disk 27-3
playing into a graphics presentation space 27-3
printing 36-9
resetting the presentation space before playing 27-3
restrictions on contents 27-5
SAA-conforming 27-6
saving on disk 27-3
sharing with other applications 27-5
use of the clipboard 27-5
uses 27-1
metric units 19-4
MIA_ values 5-10
mice, hardware system limits G-2
micro presentation space 3-19, 4-8
calls valid in B-1
migrating applications to the Presentation Manager 40-1
MIS_ values 5-10
mix attribute 21-1, 21-6
background 21-7
foreground 21-6
leave-alone mix 21-9
OR mix 21-8
overpaint 21-7
specifying foreground and background 21-9
XOR mix 21-8
MKMSGF 39-2
mnemonics 5-13
MO-DCA graphics interchange standard 27-1
modal dialog 7-4
modal dialog box 7-2
modal interface 18-1
modal message box 7-20
model space 25-2
model transformation 25-2, 25-3
modeless dialog 7-2
modeless dialog box 7-2
module definition file statement, NEWFILES 33-8
module definition file, NEWFILES 33-27
monocasing 39-11
monochrome devices, color graphics 21-9
Mou Calls 1-7
MouClose 31-5
MouDeRegister 31-8
MouDrawPtr 31-7
MouGetDevStatus 31-6
MouGetEventMask 31-6
MouGetNumButtons 31-6
MouGetNumMickeys 31-6
Mou Calls *(continued)*
MouGetNumQueEl 31-7
MouGetPtrPos 31-7
MouGetPtrShape 31-8
MouGetScaleFact 31-6
MouOpen 31-5
MouReadEventQue 31-6
MouRegister 31-8
MouRemovePtr 31-7
MouSetDevStatus 31-6
MouSetEventMask 31-6
MouSetPtrPos 31-7
MouSetPtrShape 31-8
MouSetScaleFact 31-6
MouSynch 31-8
MOUSE device 33-10
mouse device, file I/O 33-1
mouse input 3-16
mouse input by text-windowed and full-screen applications
closing the mouse 31-5
defining the pointer shape 31-8
determining buttons supported 31-6
opening the event queue 31-5
querying elements in the queue 31-7
querying mickeys per centimeter 31-6
reading a record from the queue 31-6
replacing mouse function calls 31-8
selecting mouse events to record 31-6
setting mouse state 31-6
setting scaling factors 31-6
setting the pointer position 31-7
synchronizing screen with mouse 31-7
mouse messages 3-4
moveable heaps 13-23
moving a file 33-13
MSGBIND 39-2
multiline entry fields 7-14
multilingual code page (CP850) 39-1, 39-3, 39-8
multiple initialization files 14-1
multiple segments, allocating 32-2
multiple segments, reallocating 32-5
multitasking 1-1

N

named pipe function calls
DosCallNmPipe 35-15
DosClose 35-13, 35-14
DosConnectNmPipe 35-13
DosDisConnectNmPipe 35-13
DosMakeNmPipe 35-12
DosOpen 35-13
DosPeekNmPipe 35-15
DosQNmPHandState 35-16
DosQNmPipeSemState 35-17
DosRead 35-14
DosReadAsync 35-14

named pipe function calls *(continued)*
DosSetNmPHandState 35-16
DosSetNmPipeSem 35-16
DosTransactNmPipe 35-15
DosWrite 35-14
DosWriteAsync 35-14
named pipes
access and sharing modes 35-14
access mode of 35-12
accessing by a client 35-13
blocking reads and writes 35-12
byte stream pipe 35-12
changing the state of 35-13
closing and deallocating 35-13
coordinating access to local pipes 35-16
creating by the server process 35-12
determining handle state 35-16
determining pipe state 35-15
ending access to 35-14
examining messages in 35-15
handles for 35-11
inheritance flag 35-12
initial state of 35-13
message pipe 35-12
multiple instances of 35-12
preparing for next client 35-13
processing transactions 35-15
querying buffer sizes 35-16
reading from 35-14
setting file write-through 35-12
specifying buffer sizes for 35-12
waiting for a client to open 35-13
writing to 35-14
named shared segment 35-10
name, FSD 33-6
naming character devices 33-10
naming conventions, EA 33-17
naming conventions, file 33-6
naming extended attributes 33-17
national language support 39-1
See also code page
national options
setting 14-1
network applications 33-6
network parameters 37-3
NEWFILES declaration, module definition file 33-27
NEWFILES module definition file statement 33-8
non-bootable HPFS 33-5
non-critical EAs 33-26, 33-27
non-8.3 file name format 33-7
nonreentrant language sample programs 11-1
DLG1COB.COB dialog box sample application 11-1
DLG1COB.FOR dialog box sample application 11-1
nonreentrant languages
limitations 3-11
messages posted on 3-11
nonreentrant languages *(continued)*
mixed-language programs 3-8, 3-12
PM program structure 3-8
window classes 7-28
nonretained graphics 22-1
Nordic code page (CP865) 39-3
normal PS 3-19, 4-8
note tag (IPF) H-43
nt tag (IPF) H-44
NUL device 33-10
nul device, file I/O 33-1

O

object window, creating 13-14
online command reference 33-2
OpenFlag parameter, DosOpen 33-11
OpenMode parameter, DosOpen 33-12
OR mix 21-8
ordered list (IPF)
See list tags (IPF)
orders, graphics D-1
origin, presentation page 19-3
origin, window 3-2
OS2SYS.INI 37-3
See also initialization files
application names 14-2
contents 14-1
OS2.INI
See also initialization files
application names 14-1
contents 14-1
OS2.INI initialization file 13-9
OS/2 API 1-3
OS/2 FAT disks, accessing 33-3
OS/2 FAT file system 33-2, 33-3
OS/2 mode 1-1
OS/2 1.2 Programming Tools 2-1
contents 2-1
OS/2-mode applications 1-4
outline fonts 23-1, 23-8
output area 37-7
OUTPUT device 33-1
overpaint mix 21-7
owner window 5-2, 7-1

P

packing messages 3-3
page viewport 19-5
painting calls 3-22
panel area, definition of X-7
panel body, definition of X-7
panel definition
definition of X-8
panel definition, IPF 17-2
panel ID, definition of X-8

paragraph tag (IPF) H-46
parallel printer 33-10
PARAM structure 13-2
parameter list tags (IPF)
 parml (parameter list) H-47
 pd (parameter description) H-49
 pt (parameter term) H-50
parameter list (IPF) H-47
parent windows 3-1
partial arc primitive 20-14
partitions 33-1, 33-2
partitions, FAT 33-5
partitions, HPFS 33-5
Pascal calling convention 1-9
password 13-3
path length, maximum 33-11
path names 33-4
paths 20-3, 20-18, 25-11, 26-1
 as clipping boundaries 26-2
 as filled areas 20-19
 as wide lines 20-19
 calls valid in B-1
 defining 20-18
pattern reference point 20-12
pattern sets 23-15
PC-DOS 1-2
PC-DOS emulation 1-4
PC-DOS interrupts and OS/2 function call equivalencies F-1
pels 19-3
physical color table 21-2
physical fonts 23-2
physical video buffer, accessing by full-screen applications 30-10
pick aperture 22-13
pick tag 22-13
PicPrint parameters 36-10
picture calls
 long names
 PicIchg 27-7
 PicPrint 36-10
 short names
 PICONV 27-7
 PIPRT 36-10
Picture Interchange Format files 27-7
PIF files 27-7
 printing 36-9
pipes
 See also named pipes, unnamed pipes
 passing data between related processes 35-7
 passing data between unrelated processes 35-11
 system limits G-3
pipes, file I/O 33-1
planes, color 21-1
PLOTTERS.DRV 37-2, 37-4
PM API 1-2
PM applications 1-4
 responsibilities of 1-2
PM limitations
 COBOL PM programs 3-11
 FORTRAN PM programs 3-11
PM messages 1-2
 extracting data from 3-3, 4-9
PMORD.H, graphics orders header file D-1
PMPRINT queue processor 36-5, 37-3
PM_Q_RAW format 36-3
 printing 36-6, 36-12
PM_Q_STD format 36-3
 printing 36-7
pointer 3-4
 editing 8-1
POINTER device 33-10
pointer device, file I/O 33-1
POKE transaction, (DDE) 29-7
polyline primitive 20-1
pop-up session, Vio 30-6
pop-up window 3-23, 9-8
ports, hardware system limits G-2
Portugese code page (CP860) 39-3
posting messages 3-7
preemptive addition 25-16
PRELOAD segment 34-15
preparing file systems for system shutdown 33-6
presentation drivers 36-3, 37-3, 38-3
 dialog box 37-4
 querying 37-5
Presentation Manager API 1-1
Presentation Manager applications 1-2
Presentation Manager limits 13-1, G-2
presentation page 19-1, 19-3
 coordinate range 19-4
 current position 19-3
 default size 19-5
 defining 19-3
 origin 19-3, 19-5
 size 19-4
 units 19-3
presentation parameters
 PP_BACKGROUNDCOLOR 13-2
 PP_BACKGROUNDCOLORINDEX 13-2
 PP_BORDERCOLOR 13-3
 PP_BORDERCOLORINDEX 13-3
 PP_DISABLEDBACKGROUNDCOLOR 13-3
 PP_DISABLEDBACKGROUNDCOLORINDEX 13-3
 PP_DISABLEDFOREGROUNDCOLOR 13-3
 PP_DISABLEDFOREGROUNDCOLORINDEX 13-3
 PP_FONTNAMESIZE 13-3
 PP_FOREGROUNDCOLOR 13-2
 PP_FOREGROUNDCOLORINDEX 13-2
 PP_HILITEBACKGROUNDCOLOR 13-2
 PP_HILITEBACKGROUNDCOLORINDEX 13-3
 PP_HILITEFOREGROUNDCOLOR 13-2
 PP_HILITEFOREGROUNDCOLORINDEX 13-3
Presentation Space 3-18, 3-19, 4-8, 19-1
 associating with device context 19-2
 cached micro 19-1, 19-6

Presentation Space *(continued)*
creating 19-7
defining 19-1
deleting 19-3
disassociating from device context 19-2
graphics 36-7
handle 19-1
micro 19-1, 19-2, 22-2, 27-2, B-1
normal 19-1
origin 19-3
purpose 19-1
reassociating 23-5
resetting 27-3
reusing 22-11
presentation space, alphanumeric
See Vio presentation space
presentation space, system limits for G-2
presentation-page space 25-4
PRESPARAMS structure 13-2
primary window 3-23
definition 3-23
definition of X-8
primitive attributes 20-1
primitives 20-1
arc 20-12
circle 20-13
ellipse 20-13
fillet 20-16
full arc 20-14
partial arc 20-14
spline 20-17
3-point arc 20-16
area 20-10
background of 21-5
box 20-2
character string 23-8
default attributes 20-1
direction of drawing 20-11
foreground of 21-5
image 20-4
line 20-1
marker 20-3
polyline 20-1
Print Manager 36-5, 37-3
Print Picture utility 36-9
printer
changing system printers 14-1
changing the default printer 14-1
printer address 36-13, 37-2, 37-3
printer device context 37-10
printer device, file I/O 33-1
printer selection 37-3
printing
alphanumeric data 36-6, 36-12
bit maps 36-11
color 37-8
copy of the screen 36-10
errors 36-12
printing *(continued)*
graphic data 36-7
graphics 36-9
graphics text 36-8
metafiles 36-9
PIF files 36-9
PM_Q_RAW data 36-12
PM_Q_STD data 36-7
raw data 36-6, 36-12
scheduling 37-3, 37-4
spooler not selected 36-13
stopping before spooling 36-12
text mode 36-8
using a separate thread 36-12, 36-14
using device calls 36-6, 36-16
using Dos calls 36-7
using spooler calls 36-12
using the Print Picture utility 36-9
using the spooler 36-1
without the spooler 36-13
priority for a thread, setting 34-10
priority level, changing process 34-11
priority of a graphics segment 22-7
priority of print jobs 37-3
private fonts 23-2
privilege levels, IOPL 34-1
PRN, 33-10
process
cleanup operations by 34-9
determining environment strings for 34-9
ending a child 34-9, 34-13
ending itself 34-9
notifying of child session ending 34-12
notifying of child session starting 34-18
setting priorities for child 34-10
setting selectability of 34-13
starting a child 34-9
starting independent 34-9
synchronous or asynchronous execution of 34-9
system limits G-3
waiting for child to end 34-9
process function calls
DosCWait 34-9
DosExecPgm 34-9
DosExit 34-9
DosExitList 34-9
DosGetPrty 34-10
DosKillProcess 34-9
DosSetPrty 34-10
process ID 32-3, 34-11
process, definition 1-1
profile calls 1-7
long names
PrfAddProgram 14-4
PrfChangeProgram 14-4
PrfCloseProfile 14-3
PrfCreateGroup 14-4
PrfDestroyGroup 14-4
PrfOpenProfile 14-2, 14-3

profile calls *(continued)*
long names *(continued)*
PrfQueryDefinition 14-4
PrfQueryProfile 14-3
PrfQueryProfileData 14-3
PrfQueryProfileSize 14-3
PrfQueryProfileString 14-3, 36-8, 37-3, 37-4, 37-10
PrfQueryProgramCategory 14-4
PrfQueryProgramHandle 14-4
PrfQueryProgramTitles 14-4
PrfRemoveProgram 14-4
PrfReset 14-3
PrfWriteProfileData 14-3
PrfWriteProfileString 14-3
WinInstStartApp 14-4
WinTerminateApp 14-4
short names
PRADPR 14-4
PRCGRP 14-4
PRCPR 14-3
PRCPRG 14-4
PRDGRP 14-4
PROPPR 14-2, 14-3
PRQDEF 14-4
PRQPC 14-4
PRQPDA 14-3
PRQPHD 14-4
PRQPR 14-3
PRQPST 14-3, 37-3, 37-4, 37-10
PRQPSZ 14-3, 36-8
PRQPTI 14-4
PRRES 14-3
PRRPRG 14-4
PRWPDA 14-3
PRWPST 14-3
WMISAP 14-4
WMSTOP 14-4
Program list
See installed program list
See running programs list
Programming Tools
Dialog Box Editor, definition of 2-13
Font Editor, definition of 2-13
Icon Editor, definition of 2-13
Resource Compiler, definition of 2-14
WINABLE, definition of 2-14
programs, sample
DIALOG1 9-1—9-9
DIALOG2 5-3—5-5, 10-1—10-22
GRAPHIC1 19-7, 22-15—22-18
HELLO1 4-1—4-12
HELLO2 6-1—6-9
IMAGE 13-10—13-14
propagate detectability attribute of graphics segments 22-8
propagate visibility attribute of graphics segments 22-8
proportional spacing, graphics fonts 23-7
pseudo-character devices 33-1
public fonts 14-2, 23-2, 36-9
public window class, registering 13-9
publications, list of vii
pull-downs 5-8
pull-down, definition of X-8
push-and-set mode, graphics attributes 22-11
pushbutton 7-8

Q

query calls, graphics 18-2
queue function calls
DosCloseQueue 35-10
DosCreateQueue 35-9
DosOpenQueue 35-9
DosPeekQueue 35-9
DosPurgeQueue 35-10
DosQueryQueue 35-10
DosReadQueue 35-9
DosWriteQueue 35-9
queue name 37-2
queue-processor name 37-3
queue-processor parameters 36-13, 37-3, 37-7
examples 37-8
queue-processor selection 37-4
queued device context 36-2, 36-6, 37-1
queues 36-1, 37-4
adding an element to 35-9
creating by a server process 35-9
definition of 35-8
deleting from memory 35-10
elements of 35-8
emptying elements from 35-10
ending access to 35-10
examining an element of 35-9
ordering of elements of 35-9
querying number of elements in 35-10
removing an element from 35-9
storing data 35-10
system limits G-3
waiting on a queue with a semaphore 35-10

R

radio buttons 7-7
RAM semaphores
definition of 35-2
Fast-Safe RAM data structure 35-2
handle for 35-2
ownership tracking of 35-3
shared access to 35-2
use count for 35-3
raster fonts 23-1, 23-8
raster operations 24-1
raw data 36-3
printing 36-6, 36-12

rcinclude statement 7-27
read-only file access 33-12
read-only file attribute, changing 33-13
Read-only files 33-12
reading a file, pipe, or device 33-31
read/write file access 33-12
read/write pointer 33-30
realization of color tables 21-4
reallocating memory 32-4
recognizing file objects 33-3
RECOVER 33-5
recovering from system limits G-1
recursive window procedures 4-9
redirection 33-2, 33-6
redrawing
 using bit maps 24-1
reentrancy 3-8
reentrant PM applications 3-6
reference phrase
 definition of X-9
reference phrase help
 definition of X-9
reflection transformation 25-13
region handle 26-4
regions
 clipping 26-1, 26-4
 combining 26-4
 defining 26-4
 updating 26-4
remote file system 33-5
removable media 33-3
renaming a file 33-13
REQUEST transaction, (DDE) 29-6
reserved device names 33-10
resizing the stack 34-1
resource compiler 6-1, 7-27, 8-5, 8-6, 8-7, 8-10
resource file (.RES) 7-27
 creating 8-6, 8-7
 dynamic linking 8-10
resource script file (.RC) 6-1
 accelerators 5-13
 BITMAP statement 8-3
 dialog template 7-27
 directives 8-6
 FONT statement 8-9
 ICON statement 8-3
 menu template 5-9
 POINTER statement 8-3
 rcinclude statement 7-27
 writing 8-5
 .DLG file 7-27
 .H file 7-27
resource segment, freeing 32-8
resources
 translating 39-1
retain-mode graphics 22-1
retained graphics 22-1
retained segment 22-1
return code, error A-1
RGB mode, color tables 21-3
RGB values 21-3
right margin (IPF) H-51
ROM cell sizes 30-2
root graphics segments 22-6
rotation matrix 25-14
rotation transformation 25-13
run-time dynamic linking 34-16
run-time dynamic linking function calls
 DosFreeModule 34-17
 DosGetModHandle 34-16
 DosGetModName 34-16
 DosGetProcAddr 34-17
 DosLoadModule 34-16
running programs list 14-4
 function calls 14-5
 querying and changing 14-4

S

SAA (Systems Application Architecture) 1-1, 1-3, 1-7, 3-22, 18-1
sample programs
 COBOL
 DLG1COB 2-12, 11-1
 DLG2COB 2-12, 12-1
 control program
 CRERR(.C and .ASM) 2-12
 DEMODD.ASM 2-13
 DLNK1(.C and .ASM) 2-12
 DLNK2(.C and .ASM) 2-12
 DLNK3(.C and .ASM) 2-12
 DYNLINK(.C and .ASM) 2-12
 INIT(.C and .ASM) 2-12
 IOPL2.ASM 2-13
 MEMORY(.C and .ASM) 2-12
 MONITOR.ASM 2-13
 MOUSE(.C and .ASM) 2-12
 PROCESS.ASM 2-13
 PROG1.ASM 2-13
 PROG2.ASM 2-13
 PROTMODE.ASM 2-13
 REALMODE.ASM 2-13
 THREAD.ASM 2-13
 VIOSAMP(.C and .ASM) 2-12
 WT(.C and .ASM) 2-12
 C/2
 AVIOSAMP 2-9
 BMAP 2-9, 24-7
 CLIPBRD 2-7, 28-3
 DIALOG1 2-4, 9-1—9-9
 DIALOG2 2-4, 10-1—10-22
 FONTTEST 2-11
 GRAPHIC1 2-6, 19-7, 22-15—22-18
 GRAPHIC2 2-6
 HELLO1 2-4, 4-1—4-12
 HELLO2 2-4, 6-1—6-9

sample programs *(continued)*
C/2 *(continued)*
IMAGE 2-8, 13-10—13-14
TEMPINIT 5-3—5-5
TEMPLATE 2-11
TYPETEXT 2-10
FORTRAN
DLG1FOR 2-12, 11-1
DLG2FOR 2-12, 11-1
graphics 18-2
Macro Assembler/2
HELLOM 2-12
saving application state 14-5
SBCS (single-byte character set) 7-13
scaling matrix 25-14
scaling transformation 25-12
scan line 24-1
screen device, file I/O 33-1
SCREEN$ 33-10
scroll bars 5-16, 7-18
scrolling 25-6
scrolling text 30-7
SEA (standard EA) 33-17
search characters 33-10
secondary window 3-23
segment 22-1
adding elements 22-3
attributes of 22-6
called from other segments 22-6, 22-14, 25-1, 25-3
chain 22-6
correlating 22-13
drawing 22-7, 22-9
chained to other segments 22-6
changing the attributes of 22-9
copying 22-5
creating 22-2, 22-3
deleting from segment store 22-5
detectable 22-8
drawing dynamic 25-3
drawing mode when created 22-1, 22-6
drawing on an output device 22-9
dynamic 22-8
dynamic, drawing 22-10
editing 22-4
elements of 22-3
example code
creating segments 22-15
fast-chaining attribute 22-7
inserting data in 22-4
naming 22-2
nonretained 22-1
pickable 22-8
position in the segment chain 22-7
propagate-detectability 22-8
propagate-visibility 22-8
renaming 22-6
replacing data in 22-4
retained 22-1
segment *(continued)*
reusable 22-1
root 22-6
store 22-1
transformation 25-3
unchained 22-6, 27-3
visible 22-8
zero 22-2, 22-4, 22-5, 22-6
segment of memory
See also memory management
allocating up to 64KB 32-1
avoiding large allocations 32-3
changing size of 32-5
child process 32-3
code segment descriptor 32-4
creating a shared segment 32-1
determining availability 32-3
discardable 32-2
discardable segments 32-4
global sharing 32-3
initializing for suballocation 32-5
limitations of suballocating 32-5
local sharing 32-3
multiple, allocating 32-2
named shared segment 35-10
process ID 32-3
reallocating 32-5
shared, creating a 32-3
size of allocated segment 32-7
suballocating small pieces 32-4
system limits G-1
64KB needed by memory 32-2
segments
calls valid in B-1
semaphore function calls
DosCloseSem 35-2
DosFSRamSemClear 35-4
DosFSRamSemRequest 35-4
DosMuxSemWait 35-6
DosOpenSem 35-2
DosSemClear 35-4
DosSemRequest 35-4
DosSemSet 35-4
DosSemSetWait 35-6
semaphores
See also RAM semaphores, system semaphores
comparison of 35-3
serializing resource use with 35-3
signaling occurrence of events with 35-4
send msg hook 13-19
sending messages 3-7
session
debugging of child 34-12
starting independent or child 34-11
switching to foreground 34-18
system limits G-2
tracing processes in a child 34-17

session function calls
DosSelectSession 34-13
DosSetSession 34-13
DosStartSession 34-11
DosStopSession 34-13
session, definition 1-1
set mode, graphics attributes 22-11
shared files 33-12
shared segments
child process 32-3
DosGiveSeg 32-3
global 32-3
local 32-3
locking/unlocking 32-4
privilege level 32-4
process ID 32-3
release for discarding 32-4
usage count 32-7
\SHAREMEM subdirectory 32-3
shared segment, creating a 32-1
sharing EAs 33-23
sharing memory 32-3
shearing 25-20
Shell 1-2
control panel 13-9
load list DLL 13-9
shell API 14-1—14-7
shift count 32-2
shutting down file systems 33-6
sibling window 3-2
signals 35-6
system limits G-3
simple list (IPF)
See list tags (IPF)
single line entry fields 7-12
Spl calls 1-8
spline primitive 20-17
spool-file format 36-3
spooler 36-1
benefits 36-1
invoking directly 36-12
not selected 36-13
overview 36-2
Print Manager 36-5
queue processor 36-5
parameters 37-7
user interface 36-5
spooler calls 1-7, 1-8, 36-12, 36-13
long names
SplQmClose 36-13
SplQmEndDoc 36-13
SplQmOpen 36-12
SplQmStartDoc 36-12
SplQmWrite 36-13
short names
SMCLS 36-13
SMEDOC 36-13
SMOPEN 36-12
SMSDOC 36-12
spooler calls *(continued)*
short names *(continued)*
SMWRT 36-13
spooler parameters 36-13, 37-3
spoolerparams 37-3
stack space, minimum 34-1
standard devices 33-1, 33-2
standard EA (SEA) 33-17
standard extended attribute, SEA 33-19
standard window 5-1
action bar 5-8
adding your own controls 5-20
client window 5-19
moving 5-7
reformatting 5-20
scroll bar 5-16
sizing 5-15
title 5-8
static controls 7-21
bit maps 7-21
frames 7-21
group box 7-21
icons 7-21
rectangles 7-21
text 7-21
stop-draw condition 18-2
storage device 33-2
string atoms, system limits G-2
string tables 8-4
structures 7-27
DLGTEMPLATE 7-27
in COBOL 11-6, 12-15, 12-35
in FORTRAN 11-7, 12-15, 12-35
structures, data
FRAMECDATA 5-6
suballocating memory 32-5
changing size of 32-6
DosSubAlloc 32-5
DosSubSet 32-5
initializing 32-5
limitations 32-5
organizing segments 32-5
size of segment 32-5
subclassing a window 3-14, 13-9
subdirectories 33-12
changing current 33-15
creating 33-13
determining current 33-15
removing 33-15
subdirectory, \SHAREMEM 32-3
SUBJECT extended attribute 33-20
suspending execution of threads 34-2
switch list
See running programs list
symbols (IPF) I-1
synchronous painting 3-20
synchronous processing, messages 3-17

SYS 33-5
SYSCLR_
 SYSCLR_APPWORKSPACE 14-7
 SYSCLR_HELPBACKGROUND 14-7
 SYSCLR_HELPHILITE 14-7
 SYSCLR_HELPTEXT 14-7
 SYSCLR_OUTPUTTEXT 14-7
 SYSCLR_WINDOW 14-7
 SYSCLR_WINDOWSTATICTEXT 14-7
 SYSCLR_WINDOWTEXT 14-7
system applications 14-1
system colors 21-5
 indexes 14-7
 querying 14-7
system file attribute, changing 33-13
System files 33-12
system files, HPFS 33-5
system font 23-1
system limits G-1
 global descriptor tables G-1
 hardware G-2
 libraries G-1
 local descriptor tables G-1
 memory resources G-1
 named shared segments G-1
 open file handles G-3
 pipes G-3
 Presentation Manager G-2
 processes G-3
 queues G-3
 recovering from G-1
 segments G-1
 sessions G-2
 system semaphores G-3
 system signals G-3
 threads G-3
 windows 3-24
system memory, available 32-3
system menu icon 5-15
system semaphores
 accessing by another process 35-2
 definition of 35-2
 exclusive 35-2
 handle for 35-2
 nonexclusive 35-2
 ownership tracking of 35-1
 system limits G-3
 use count for 35-2
system topic 29-10
system values
 querying 14-5
Systems Application Architecture (SAA) 1-1, 1-3, 1-7, 3-22, 18-1

T

tag language markup (IPF)
 See also Information Presentation Facility
tag language markup (IPF) *(continued)*
 bit map graphics 17-12
 character graphics 17-11
 figure captions 17-11
 figures 17-11
 headings 17-2
 highlighted phrases 17-6
 linking
 from a hypergraphic area 17-17
 from a hypertext phrase 17-13
 link definition file 17-17
 to a smaller window 17-14
 to another panel 17-13
 to send the application a message 17-14
 to start another application 17-16
 lists 17-7
 definition 17-9
 nesting 17-9
 ordered 17-8
 parameter 17-10
 simple 17-7
 unordered 17-8
 notes 17-6
 notices 17-7
 special characters 17-12
 unformatted lines
 in figures 17-11
 monospace font 17-11
 proportional space font 17-12
tagging help panels 17-1
tagging IPF help panels 17-2
tag, definition of X-10
Task List 34-11
 See also running programs list
TEMPINIT sample application 5-3—5-5
template, dialog 7-26
terminating a PM application 4-7
text alignment 23-13
text formatting 23-13
text message file for translation 39-2
text mode 36-8
text-windowed applications 1-2, 1-3, 1-4
text, graphics 23-1
thread
 definition 1-1
 ending execution of 34-3
 minimum stack space 34-1
 setting priority for 34-10
 starting by application 34-1
 starting by OS/2 34-1
 starting from IOPL code segment 34-1
 suspending all within process 34-3
 suspending one within process 34-3
 system limits G-3
thread for printing 36-12, 36-14
thread function calls
 DosCreateThread 34-1
 DosEnterCritSec 34-3

thread function calls *(continued)*
DosExit 34-3
DosExitCritSec 34-3
DosGetPrty 34-10
DosResumeThread 34-3
DosSetPrty 34-10
DosSuspendThread 34-3
timer
determining duration of clock tick for 34-14
starting asynchronous 34-14
starting asynchronous recurring 34-14
starting synchronous 34-14
stopping 34-14
timer function calls
DosSleep 34-14
DosTimerAsync 34-14
DosTimerStart 34-14
DosTimerStop 34-14
title bar 5-7
token parameter 37-1
TOOLKT12 subdirectory 8-8
transformation 19-4, 25-1
accumulating 25-16
concatenating 25-17
current 25-1
default viewing 25-4, 25-6
device 19-5, 25-8, 25-19
equations 25-11
identity 25-1, 25-15
instance 25-3
linear equations 25-11
matrixes 25-14
coding on GPI calls 25-15
model 25-2, 25-3
of bit-map data 25-10
reflection 25-13
rotation 25-1, 25-13
scaling 25-1, 25-12
segment 25-3
shearing 25-20
translation 25-1, 25-11
unity 25-1
viewing 25-4, 25-5
windowing-system 25-9
translating
full-screen-application messages 39-2
Presentation Manager messages 39-1
translation matrix 25-14
translation transformation 25-11
trap events 35-6
twips 19-4, 23-1
type definition, fonts 8-8
TYPE extended attribute 33-19
type of device 33-6
typefaces 8-8

U

UHPFS.DLL, HPFS file system utilities 33-5
unchained graphics segments 22-6
unit circle 20-13
unity transformation 25-1
unlocking discardable memory 32-4
unlocking/locking shared segments 32-4
unnamed pipe function calls
DosClose 35-8
DosDupHandle 35-7
DosMakePipe 35-8
DosRead 35-8
DosReadAsync 35-8
DosWrite 35-8
DosWriteAsync 35-8
unnamed pipes
adding data to 35-8
blocking of writes to 35-8
creating 35-8
definition of 35-7
deleting from memory 35-8
handles for 35-8
maximum size of 35-7
removing data from 35-8
unordered list (IPF)
See list tags (IPF)
unpacking messages 3-3
update region 3-19
usage count 32-7
user document (IPF) H-57
user interface 14-1
querying and changing the interface 14-1—14-7
user shell 1-2, 1-3
control panel 8-9
user-driven data exchange 28-1
See also clipboard
U.S. code page (CP437) 39-3

V

value, extended attribute 33-17
vector font 23-1, 23-8
VERSION extended attribute 33-20
video buffer, physical 30-10
video subsystem support
See also alphanumeric video output
for all application types 30-3
full-screen applications only 30-9
PM applications only 30-1
video subsystem, replacing by full-screen applications 30-12
viewing pipeline 25-1, 25-10, 26-1
viewing transformation 25-4, 25-5
viewing window 25-5, 26-1, 26-2
viewport 19-5
Vio calls 1-7
VioAssociate 30-2

Vio calls *(continued)*
VioCreateLogFont 30-2, 39-7
VioCreatePS 30-1
VioDeRegister 30-12
VioDestroyPS 30-1
VioEndPopUp 30-6
VioGetAnsi 30-5
VioGetBuf 30-3
VioGetConfig 30-5
VioGetCurPos 30-6
VioGetDeviceCellSize 30-2
VioGetFont 30-9
VioGetMode 30-7
VioGetPhysBuf 30-11
VioGetState 30-9
VioGlobalReg 30-13
VioModeUndo 30-12
VioModeWait 30-11
VioPopUp 30-6
VioQueryFonts 30-2
VioReadCellStr 30-5
VioReadCharStr 30-5
VioRegister 30-12
VioSavRedrawUndo 30-12
VioSavRedrawWait 30-11
VioScrLock 30-11
VioScrollDn 30-7
VioScrollLf 30-7
VioScrollRt 30-7
VioScrollUp 30-7
VioScrUnlock 30-11
VioSetAnsi 30-5
VioSetCp 30-5, 39-3, 39-5
VioSetCurPos 30-6
VioSetCurType 30-6
VioSetDeviceCellSize 30-2
VioSetFont 30-10
VioSetMode 30-7
VioSetOrg 30-2
VioSetState 30-9
VioShowBuf 30-3
VioWrtCharStr 30-4
VioWrtCharStrAtt 30-5
VioWrtNAtt 30-5
VioWrtNCell 30-5
VioWrtNChar 30-5
VioWrtTTY 30-4
VIO cell sizes 30-2
Vio handle 30-13
Vio presentation space 30-1
aligning a cell of 30-2
associating the window with 30-2
creating 30-1
defining font requirements 30-2
format of cell 30-1
setting the cell size of 30-2
VioGlobalReg 30-13
virtual keys 3-4
visibility attribute of graphics segments 22-8
visible region 3-19
VK_ virtual key values 3-4
volume 33-2

W

warning tag (IPF) H-58
WC_, see window class 3-13
WDOPEN parameters 37-1
well-behaved applications 1-2
width limit for cursor G-2
WINABLE utility 40-2
winding mode, area primitive 20-11
window
active 3-5
class
See also window class
without a dialog box 7-28
client 6-1
coordinates 3-2
creating an object window 13-14
creation 3-14, 4-5
definition of 3-1, X-11
dialog box 3-24
frame 6-1
handle 3-3
hierarchy 3-1
input queue 3-6
invalidation 3-20
origin 3-2
ownership 5-2
painting calls 3-22
parent-child relationship 3-1
pop-up 3-23
position 3-2, 4-6
primary 3-23
procedure 3-8, 4-7
COBOL 3-8
FORTRAN 3-8
programming without window procedures 3-8
secondary 3-23
sibling 3-2
subclassing 3-14
system limit 3-24
viewing order 3-2
visible region 3-19
words 4-5, 13-14
window calls
long names
WinAddAtom 13-22
WinAddSwitchEntry 14-5
WinAlarm 7-19
WinAllocMem 13-22
WinAvailMem 13-24
WinBeginPaint 3-19, 3-21, 3-22, 4-8, 6-9, 19-6
WinCallMsgFilter 13-19
WinChangeSwitchEntry 14-5

window calls *(continued)*
long names *(continued)*
WinCloseClipbrd 28-4, 28-6
WinCpTranslateChar 39-7
WinCpTranslateString 39-7
WinCreateAtomTable 13-22
WinCreateDataStructure 12-38
WinCreateDlg 7-3, 7-5, 7-27
WinCreateHeap 13-22
WinCreateMenu 5-13
WinCreateMsgQueue 3-6, 3-7, 3-17, 4-5, 13-9
WinCreatePointer 8-3
WinCreateStdWindow 3-14, 4-5, 5-1, 5-3, 6-7, 8-9
WinCreateWindow 3-14, 5-1, 5-3, 6-7, 7-28
WinDdeInitiate 29-3, 29-5
WinDdePostMsg 29-3, 29-5
WinDdeRespond 29-3, 29-5
WinDefDlgProc 7-1, 9-8
WinDefWindowProc 4-9, 14-5
WinDeleteAtom 13-22
WinDestroyAtomTable 13-22
WinDestroyDataStructure 12-38
WinDestroyMsgQueue 3-6, 3-7, 4-7
WinDestroyWindow 4-7
WinDismissDlg 7-5
WinDispatchMsg 3-6, 3-7, 5-11
WinDlgBox 7-3, 7-5, 8-10, 9-8
WinDrawBitmap 2-9, 24-3
WinDrawText 23-8
WinEmptyClipbrd 28-6
WinEnableWindow 7-7
WinEnableWindowUpdate 5-12
WinEndPaint 3-21, 3-22, 4-8, 6-9, 19-6
WinEnumClipbrdFmts 28-6
WinExcludeUpdateRegion 3-22
WinFillRect 4-8, 9-8
WinFindAtom 13-22
WinFreeMem 13-22
WinGetKeyState 13-18
WinGetMsg 3-6, 3-7, 3-18, 4-7, 4-9
WinGetPS 3-22, 19-6
WinGetScreenPS 3-22, 19-6, 36-10
WinInitialize 4-5
WinInvalidateRect 3-20, 3-22, 4-8
WinInvalidateRegion 3-20, 3-22, 6-8
WinIsRectEmpty 28-3
WinIsWindowShowing 3-21
WinLoadAcceltable 8-10
WinLoadDialog 7-3
WinLoadDlg 8-10
WinLoadLibrary 8-10, 13-9
WinLoadMenu 7-27, 8-10
WinLoadPointer 8-3, 8-10
WinLoadProcedure 13-9
WinLoadString 6-8, 7-19, 8-4, 8-10
WinLockHeap 13-24
WinLockVisRegions 3-22
WinMessageBox 7-19

window calls *(continued)*
long names *(continued)*
WinModifyDataStructure 12-38
WinOpenClipbrd 28-4, 28-6
WinOpenWindowDC 3-22
WinPeekMsg 3-6, 3-7
WinPostMsg 3-7, 4-9, 5-3
WinPostQueueMsg 3-7
WinProcessDlg 7-3, 7-5
WinQueryAtomLength 13-22
WinQueryAtomName 13-22
WinQueryAtomUsage 13-22
WinQueryClassInfo 13-9
WinQueryClipbrdData 28-6
WinQueryClipbrdFmtInfo 28-6
WinQueryClipbrdOwner 28-6
WinQueryClipbrdViewer 28-6
WinQueryCpList 23-6, 39-6
WinQueryDataStructure 12-38
WinQueryDlgItemShort 10-7
WinQueryPresParam 13-3
WinQueryProfileString 13-9, 23-2
WinQuerySessionTitle 14-5
WinQuerySwitchEntry 14-5
WinQuerySwitchHandle 14-5
WinQuerySwitchList 14-5
WinQuerySysColor 14-7, 21-5
WinQuerySysPointer 3-4
WinQuerySysValue 14-5, 19-4
WinQueryTaskSizePos 14-5
WinQueryUpdateRect 3-20, 3-22
WinQueryUpdateRegion 3-20, 3-22, 26-6
WinQueryWindowPtr 6-8
WinQueryWindowText 7-14, 10-7, 13-18
WinQueryWindowULong 6-8, 13-18
WinQueryWindowUShort 6-8, 13-18
WinReallocMem 13-22
WinRegisterClass 3-13, 4-5, 13-9, 13-18
WinRegisterUserDatatype 13-4
WinRegisterUserMessage 13-4
WinReleaseHook 13-19
WinReleasePS 3-22, 19-6
WinRemovePresParam 13-3
WinRemoveSwitchEntry 14-5
WinScrollWindow 5-19
WinSendDlgItemMsg 7-6, 7-12
WinSendMsg 3-7, 5-2, 13-19
WinSetAccelTable 5-14
WinSetActiveWindow 4-6, 4-7
WinSetCapture 3-17
WinSetClipbrdData 28-4, 28-6
WinSetClipbrdOwner 28-6
WinSetClipbrdViewer 28-6
WinSetDlgItemText 7-6
WinSetFocus 4-8, 10-7
WinSetHook 13-19
WinSetKeyboardStateTable 13-18
WinSetMultWindowPos 5-20
WinSetPointer 3-4, 28-5

window calls *(continued)*
long names *(continued)*
WinSetPresParam 13-3
WinSetRect 28-3
WinSetSysValue 14-5
WinSetWindowPos 4-6, 6-7
WinSetWindowPtr 6-8
WinSetWindowText 5-8, 7-6
WinSetWindowULong 6-8, 13-18
WinSetWindowUShort 6-8, 13-18
WinShowTrackRect 3-17
WinShowWindow 6-7, 7-3, 7-7
WinStartTimer 3-17, 14-2
WinStopTimer 14-2
WinSubclassWindow 13-10
WinSubstituteStrings 7-19, 8-4
WinSwitchtoProgram 14-5
WinTerminate 4-7
WinTrackRect 3-17
WinUpdateWindow 3-22
WinValidateRect 3-20, 3-22
WinValidateRegion 3-20, 3-22
WinWindowFromID 5-3, 9-9, 10-7
WinWriteProfileString 13-9
short names
WCCRD 7-3, 7-5, 7-27
WCDDP 7-1, 9-8
WCDELD 7-5
WCLD 7-3, 8-10
WCLM 7-27, 8-10
WCLMD 7-3, 7-5, 8-10, 9-8
WCMBOX 7-19
WCPD 7-3, 7-5
WCPQM 3-7
WCQDIS 10-7
WCSDIM 7-6, 7-7, 7-12
WIBPNT 3-19, 3-21, 3-22, 4-8, 6-9, 19-6
WICCBD 28-4, 28-6
WICPOI 8-3
WICRT 3-14, 5-1, 5-3, 6-7, 7-28
WICRTS 3-14, 4-5, 5-1, 5-3, 6-7, 8-9
WIDBMP 2-9, 24-3
WIDEL 4-7
WIDTXT 23-8
WIECB 28-6
WIECF 28-6
WIEPNT 3-21, 3-22, 4-8, 6-9, 19-6
WIEUPR 3-22
WIEWU 5-12
WIFRCT 4-8, 9-8
WILKVR 3-22
WINVR 3-20, 3-22, 4-8
WINVRG 3-20, 3-22, 6-8
WIOPCB 28-4, 28-6
WIOPDC 3-22
WIQCBD 28-6
WIQCBF 28-6
WIQCBO 28-6
WIQCBV 28-6

window calls *(continued)*
short names *(continued)*
WIQHID 5-3, 9-9, 10-7
WIQSCO 14-7, 21-5
WIQSHO 3-21
WIQTXT 7-14, 10-7, 13-18
WIQURC 3-20, 3-22
WIQURG 3-20, 3-22, 26-6
WIQWPT 6-8
WIQWUL 6-8, 13-18
WIQWUS 6-8, 13-18
WIREMP 28-3
WISCBD 28-4, 28-6
WISCBO 28-6
WISCBV 28-6
WISCRL 5-19
WISENA 7-7
WISHOW 6-7, 7-3
WISMPS 5-20
WISPOS 4-6, 6-7
WISREC 28-3
WISTAT 14-2
WISTOT 14-2
WISTR 3-17
WISTXT 5-8, 7-6
WISUBW 13-10
WISWPT 6-8
WISWUL 6-8, 13-18
WISWUS 6-8, 13-18
WITREC 3-17
WIUPDT 3-22
WIVREC 3-20, 3-22
WIVRGN 3-20, 3-22
WMALRM 7-19
WMCATT 13-22
WMCMFI 13-19
WMCMQ 3-6, 3-7, 3-17, 4-5, 13-9
WMCSWE 14-5
WMDATT 13-22
WMDEIN 29-3, 29-5
WMDELA 13-22
WMDEPO 29-3, 29-5
WMDERE 29-3, 29-5
WMDISP 3-6, 3-7, 5-11
WMDMQ 3-6, 3-7, 4-7
WMDWP 4-9, 14-5
WMFNAT 13-22
WMGET 3-6, 3-7, 3-18, 4-7, 4-9
WMINIT 4-5
WMKEYS 13-18
WMLACT 8-10
WMLLIB 8-10, 13-9
WMLPOI 8-3, 8-10
WMLPR 13-9
WMLSTR 6-8, 7-19, 8-4, 8-10
WMPEEK 3-6, 3-7
WMPOST 3-7, 4-9, 5-3
WMQALN 13-22
WMQANM 13-22

window calls *(continued)*
short names *(continued)*
WMQAUS 13-22
WMQCLI 13-9
WMQCPL 23-6, 39-6
WMQPST 13-9, 23-2
WMQSEN 14-5
WMQSHD 14-5
WMQSLI 14-5
WMQSPT 3-4
WMQSTI 14-5
WMQSYS 14-5, 19-4
WMQTSP 14-5
WMRCL 3-13, 4-5, 13-9, 13-18
WMRHK 13-19
WMRSWE 14-5
WMRUD 13-4
WMRUM 13-4
WMSACC 5-14
WMSACT 4-6, 4-7
WMSCAP 3-17
WMSDIT 7-6
WMSEND 3-7, 5-2, 13-19
WMSFOC 4-8, 10-7
WMSHOO 13-19
WMSKST 13-18
WMSPTR 3-4, 28-5
WMSSYS 14-5
WMSTP 14-5
WMTERM 4-7
WMWPST 13-9
WPGPS 3-22, 19-6
WPGSPS 3-22, 19-6, 36-10
WPREPS 3-22, 19-6
WUADAT 13-22
WUCDS 12-38
WUDDS 12-38
WUMDS 12-38
WUQDS 12-38
WUSUBS 7-19, 8-4
WUTRCH 39-7
WUTRST 39-7
WinAssociateHelpInstance 16-2
WinCreateHelpInstance 16-1
WinCreateHelpTable 16-10
WinDestroyHelpInstance 16-2
WinLoadHelpTable 16-11
WinQueryHelpInstance 16-13
window class
class style
CS_CLIPCHILDREN 3-14
CS_CLIPSIBLINGS 3-14
CS_FRAME 3-14
CS_HITTEST 3-14
CS_MOVENOTIFY 3-14
CS_PARENTCLIP 3-14
CS_PUBLIC 3-14
CS_SAVEBITS 3-14
CS_SIZEREDRAW 3-14
window class *(continued)*
class style *(continued)*
CS_SYNCPAINT 3-14
control 3-13
definition of 3-1
public 3-13
registering 4-5
registering a public window class 13-9
subclassing 13-9
WC_BUTTON 3-13, 7-7, 7-8, 7-9
WC_COMBOBOX 7-17
WC_ENTRYFIELD 3-13, 7-12
WC_FRAME 3-13
WC_LISTBOX 3-13, 7-10
WC_MENU 3-13
WC_MLE 7-14
WC_SCROLLBAR 3-13, 7-18
WC_SIZEBORDER 3-13
WC_STATIC 3-13
WC_TITLEBAR 3-13
window style
WS_CLIPCHILDREN 3-15
WS_CLIPSIBLINGS 3-15
WS_DISABLED 3-15
WS_GROUP 3-15
WS_MAXIMIZED 3-15
WS_MINIMIZED 3-15
WS_PARENTCLIP 3-15
WS_SAVEBITS 3-15
WS_SYNCPAINT 3-15
WS_TABSTOP 3-15
WS_VISIBLE 3-15
windowed applications 1-2
windowing-system transformation 25-9
world coordinates 19-3
world-coordinate space 25-1
limits 19-4, 25-9
write-only file access 33-12
write-through flag 33-12
writing to a file, pipe, or device 33-31
WS_, see window style 3-15

X

XFM parameter 37-7
XOR mix 21-8

Z

zero segments, graphics 22-2
zooming 25-6

Numerics

3-point arc primitive 20-16
640KB barrier 1-6
80286 machine exception error A-1

8.3 file name format 33-7

Special Characters

.DLL 8-10
.EXE file
 binding resources to 8-6
.FNT file 8-9
.H file 8-6
.RC file 5-9
.RES file 8-6
 output by Dialog Box Editor 7-27
... (ellipsis) 3-24
\SHAREMEM\ 32-3

IBM United Kingdom
International Products Limited
PO Box 41, North Harbour
Portsmouth, PO6 3AU
England

Printed in Denmark by Bonde's

